

DECnet-DOS

digital

Programmer's Reference Manual

Programmer's Reference Manual

Order Number: AA-PAFJA-TK

DECnet-DOS

Programmer's Reference Manual

November 1989

This manual documents the programming interfaces and language library provided in the DECnet-DOS kit. It details the network programming calls used in the creation of DECnet-DOS application programs.

Supersession/Update Information: This is a new manual.

Operating System and Version: MS-DOS V3.1
MS-DOS V3.2
MS-DOS V3.3
MS-DOS V4.0

Software Version: DECnet-DOS V3.0

digital™

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may only be used or copied in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital or its affiliated companies.

Copyright © 1989 by Digital Equipment Corporation
All Rights Reserved.
Printed in U.S.A.

The postage-prepaid Reader's Comments form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DECnet-DOS	VAX
DECmate	DECUS	VAXcluster
DECnet	ThinWire	VMS
digital ™	UNIBUS	VT

IBM is a registered trademark of International Business Machines Corporation.
PC/XT and Personal Computer AT are trademarks of International Business Machines Corporation.
MS, MS-DOS, and MS OS/2 are registered trademarks of Microsoft Corporation.
3Com is a trademark of 3Com Corporation.
MICOM-InterLan is a registered trademark of MICOM Systems, Inc.

Contents

Preface

Part I

1 Introduction to Network Programming

1.1	What Is DECnet?	1-2
1.1.1	How DECnet Components Communicate	1-2
1.2	What Is DECnet-DOS?	1-4
1.2.1	DECnet-DOS Components Work with Your Operating System	1-6
1.2.2	DECnet Runs As a Background Task	1-6
1.3	Which Programming Interface Should You Use?	1-7
1.3.1	Task-to-Task Communications	1-8
1.3.1.1	Transparent and Nontransparent Task-to-Task	1-9
1.3.2	Resident Interfaces	1-9
1.3.2.1	Transparent File Access (TFA) and Transparent Task-to-Task (TTT)	1-10
1.3.2.2	C Language Task-to-Task Interface	1-10
1.3.2.3	NETBIOS Emulation	1-11
1.3.2.4	Assembly Language Task-to-Task Interface	1-11
1.3.2.5	Command Terminal (CTERM) Interface	1-12
1.3.2.6	Local Area Transport (LAT) Interface	1-12
1.4	Network Programming Concepts and Considerations	1-12

1.4.1	The Client-Server Model	1-12
1.4.1.1	Client and Server Tasks	1-13
1.4.1.2	Sockets	1-13
1.4.2	How Tasks Communicate	1-15
1.4.2.1	Establishing a Logical Link	1-15
1.4.2.2	Blocking and Nonblocking I/O Operations	1-18
1.4.2.3	Node Names and Addresses	1-18
1.4.2.4	Network Object Numbers and Task Names	1-18
1.4.2.5	Access Control Information	1-18
1.4.2.6	Optional User Data	1-19
1.4.2.7	Sending Normal Data Messages	1-19
1.4.2.8	Receiving Normal Data Messages	1-20
1.4.2.9	Out-of-Band Messages	1-21
1.4.2.10	Terminating Network Activity and Closing the Logical Link	1-21

2 Transparent File Access (TFA) Interface

2.1	Introduction	2-1
2.2	TFA Command Line	2-2
2.3	Accessing Remote Files	2-2
2.4	File Characteristics	2-3
2.4.1	File Attributes	2-4
2.5	Performing Data Conversions	2-5
2.6	Converting Remote Input Files	2-6
2.6.1	Binary Image Files	2-6
2.6.2	ASCII Files	2-7
2.7	Converting Remote Output Files	2-7
2.7.1	ASCII Files	2-7
2.7.2	Image Files	2-8
2.8	Accessing TFA Services	2-8
2.8.1	Network File Specification Syntax	2-8
2.9	Issuing Function Requests	2-10
2.10	TFA Programming Considerations	2-12
	3EH Close a File Handle	2-13

3CH Create a File	2-15
41H Delete a File	2-17
4EH Find First Matching File	2-18
4FH Find Next Matching File	2-20
4BH Load and Execute a Program	2-22
3DH Open a File	2-24
3FH Read From a File/Device	2-26
40H Write to a Remote File	2-28

3 Transparent Task-to-Task (TTT) Programming Interface

3.1	Transparent Task-to-Task Communication	3-1
3.2	Transparent Communication Functions	3-1
3.2.1	Initiating a Logical Link Connection	3-2
3.2.2	Handshaking Sequence for a Client Task	3-2
3.2.3	Handshaking Sequence for a Server Task	3-2
3.2.4	Exchanging Data Messages over a Logical Link	3-3
3.2.5	Terminating the Logical Link	3-3
3.3	Command Line for Transparent Communication Tasks	3-3
3.4	Creating a Transparent Communication Task	3-4
3.5	Network Task Specifications	3-4
3.5.1	Node Specifications	3-5
3.5.2	Task Specifications	3-6
3.6	MS-DOS Intercept Routine	3-7
3.7	Using the Transparent Network Task Control Utility	3-7
3.8	TTT Programming Considerations	3-7
3.9	MS-DOS Function Requests for Transparent Task-to-Task Communication	3-8
	3EH Close File Handle	3-9
	3CH Create/Open a File	3-10
	3FH Read from a File	3-12
	Function 40H Write to a File	3-14

Part II

4 C Language Task-to-Task Interface

4.1	Creating the DECnet-DOS Programming Interface Library . . .	4-1
4.1.1	DECnet-DOS Programming Considerations	4-2
4.2	How to Read the Socket Interface Call Descriptions	4-4
4.3	Understanding a SYNTAX Section	4-5
4.4	Socket Function Calls	4-5
4.4.1	Example Socket Interface Calling Sequence	4-7
	accept	4-8
	bind	4-11
	connect	4-14
	getpeername	4-17
	getsockname	4-19
	getsockopt	4-21
	listen	4-27
	recv	4-29
	sclose	4-33
	select	4-35
	send	4-39
	setsockopt	4-43
	shutdown	4-51
	siocli	4-53
	socket	4-55
	sread	4-57
	swrite	4-60

5 C Language Subroutines

5.1	Creating the DECnet-DOS Programming Interface Library . . .	5-1
5.1.1	DECnet-DOS Programming Considerations	5-2
5.2	DECnet Utility Function Calls	5-4
	bcmp	5-6

bcopy	5-7
bzero	5-8
dnet_addr	5-9
dnet_conn	5-11
dnet_eof	5-16
dnet_getacc	5-17
dnet_getalias	5-19
dnet_htoa	5-20
dnet_installed	5-21
dnet_ntoa	5-23
dnet_otoa	5-24
dnet_path	5-25
getnodeadd	5-29
getnodeent	5-30
getnodename	5-33
nerror	5-34
perror	5-35

6 Assembly Language Task-to-Task Interface

6.1	Introduction	6-1
6.1.1	How Applications Communicate	6-2
6.1.2	Establishing Sessions	6-3
6.2	Accessing DECnet-DOS Network Process Services	6-3
6.2.1	DECnet Network Process Installation Check	6-4
6.2.2	Calling the Network Process	6-5
6.3	The Network I/O Control Block (NIOCB)	6-6
6.4	Programming Tips and Considerations	6-10
6.4.1	Handling Network I/O	6-10
6.4.1.1	Blocking Synchronous Mode	6-10
6.4.1.2	Nonblocking Synchronous Mode	6-11
6.4.1.3	Asynchronous Mode	6-11
6.4.2	Setting the io_flags Field	6-12
6.4.3	Using Socket Numbers with Network Process Interface Calls	6-14
6.5	Network Process Interface Calls	6-14

ABORT	6-17
ACCEPT	6-19
ATTACH	6-23
BIND	6-26
CANCEL	6-29
CONNECT	6-31
DETACH	6-35
DISCONNECT	6-37
GETSOCKOPT	6-39
LISTEN	6-46
LOCALINFO	6-48
PEERADDR	6-51
RCVD	6-53
RCVOOB	6-57
SELECT	6-60
SEND	6-64
SENDOOB	6-67
SETSOCKOPT	6-69
SHUTDOWN	6-77
SIOCTL	6-79
SOCKADDR	6-82

7 NETBIOS Emulation Interface

7.1	Introduction	7-1
7.1.1	LAN Communications	7-4
7.1.2	WAN Communications	7-4
7.1.3	Communication to VAX Computers	7-5
7.2	How Applications Communicate	7-5
7.3	NETBIOS Command Line Switches	7-7
7.4	Accessing NETBIOS Services	7-8
7.4.1	NETBIOS Installation Checks	7-9
7.4.1.1	Checking 2A Installation	7-9
7.4.1.2	Checking 5CH Installation	7-10
7.4.2	Issuing NETBIOS Calls	7-11

7.5	The Network Control Block (NCB)	7-12
7.6	Network Control Block (NCB) Commands	7-14
7.6.1	Using Asynchronous and Synchronous Modes	7-15
7.6.2	Return Codes for NCB Commands	7-18
7.7	Accessing Digital Extended Functions	7-18
7.8	NCB and DSCB Commands	7-22
7.8.1	NCB Commands	7-22
	ADAPTER STATUS	7-23
	ADD GROUP NAME	7-27
	ADD NAME	7-29
	CALL	7-31
	CANCEL	7-33
	CHAIN SEND	7-35
	DELETE NAME	7-37
	FIND NAME	7-39
	HANG UP	7-41
	LISTEN	7-43
	RECEIVE	7-45
	RECEIVE ANY	7-47
	RECEIVE BROADCAST DATAGRAM	7-49
	RECEIVE DATAGRAM	7-51
	RESET	7-53
	SEND	7-55
	SEND BROADCAST DATAGRAM	7-57
	SEND DATAGRAM	7-59
	SESSION STATUS	7-61
7.8.2	DSCB Commands	7-63
	ADD REMOTE NAME	7-64
	ADD SERVER NAME	7-65
	DELETE ALL REMOTE NAMES	7-66
	DELETE ALL SERVER NAMES	7-67
	DELETE REMOTE NAME	7-68
	DELETE SERVER GIVEN NODE NAME	7-69
	DELETE SERVER GIVEN NODE NUMBER	7-70
	READ ALL REMOTE NAMES	7-71

READ NUMBER OF REMOTE NAMES	7-73
INSTALLATION CHECK	7-74
READ REMOTE NAME	7-75
READ SERVER BY INDEX	7-76
READ SERVER GIVEN NODE NAME	7-77
READ SERVER GIVEN NODE NUMBER	7-78

8 Command Terminal (CTERM) Interface

8.1	Introduction	8-1
8.2	CTERM Services	8-2
8.3	CTERM Command Line	8-2
8.4	CTERM Sessions	8-3
8.4.1	Specifying Nodes to the CTERM Driver	8-4
8.4.2	Issuing Calls to the CTERM Driver	8-4
8.4.3	Exchanging Data	8-5
8.4.4	Closing a CTERM Session	8-5
8.5	Sending Calls to the CTERM Driver	8-5
8.6	CTERM Functions	8-6
	00H Installation Check	8-8
	01H Send Char	8-9
	02H Read Char	8-10
	03H Status	8-11
	04H DECnet Status	8-12
	05H Open Session	8-13
	06H Close Session	8-14
	0AH Get SCB Size	8-15
	0BH Get DECnet Socket	8-16
	0FH Deinstall CTERM	8-17
8.7	Sample Terminal Programs	8-18
8.7.1	Assembler Language Program	8-18
8.7.2	C Language Sample Program	8-24

9 Local Area Transport (LAT) Interface

9.1	Introduction	9-1
9.2	LAT Services	9-2
9.2.1	LAT Command Line	9-2
9.2.2	Service Directory	9-4
9.2.3	LAT Sessions	9-5
9.2.4	Starting a Session	9-5
9.2.5	Exchanging Data	9-6
9.2.6	Flow Control	9-6
9.2.7	LAT Call-Back Routines	9-6
9.2.8	Closing the LAT Session	9-7
9.3	Data Structures	9-7
9.3.1	The LAT Session Control Block	9-7
9.3.2	Session Status Word Definition	9-11
9.4	LAT Functions	9-12
	03H LAT Get Status	9-13
	D0H Open Session	9-14
	D0H Close LAT Session	9-15
	02H Read Char	9-16
	01H Send Char	9-17
	D5H Get Next LAT Service Name	9-18
	D3H Reset LAT Counters	9-19
	D6H LAT Service Reset	9-20
	D4H Copy LAT Counters	9-21
	D1H Send Break Signal	9-22
9.5	Sample Terminal Programs	9-23
9.5.1	Assembly Language Terminal Program	9-23
9.5.2	C Language Terminal Program	9-31

Part III

A Socket Definitions

A.1	Communications Domain	A-1
A.2	DECnet Layers	A-2
A.3	DECnet Objects	A-2
A.4	DECnet Options	A-3
A.5	Flag Options	A-6
A.6	Logical Link States	A-7
A.7	Maximum Number of Incoming Connection Requests	A-7
A.8	Socket Interface Options	A-8
A.9	Socket Types	A-9
A.10	Defined Software Modules	A-9

B Defined Data Structures and Data Members

B.1	Access Control Information Data Structure	B-2
B.2	Attach Data Structure	B-3
B.3	DECnet Node Address Data Structure	B-4
B.4	Listen Data Structure	B-4
B.5	Local Node Information Data Structure	B-5
B.6	Logical Link Information Data Structure	B-6
B.7	Optional User Data Structure	B-6
B.8	Select Data Structure	B-7
B.9	Shutdown Data Structure	B-8
B.10	Socket Address Data Structure	B-8
B.11	Socket I/O Status Data Structure	B-9
B.12	Socket Option Data Structure	B-9
B.13	User Access Control Information Data Structure	B-10
B.14	User-Defined Callback Routine Data Structure	B-11
B.15	User-Defined Buffer Data Structure	B-11

C Summary of Error Completion Codes

D Summary of Extended Error Codes

E Using the Transparent Network Task Control Utility

E.1	Displaying Status	E-1
E.2	Removing TFA and TTT From Memory	E-2
E.3	On-Line Help	E-3

F Transparent File Access Error Messages

G Data Access Protocol (DAP) Error Messages

G.1	Overview	G-1
G.1.1	Maccode Field	G-1
G.1.2	Miccode Field	G-2

H Transporting DECnet-DOS Programs

I DECnet-DOS Programming Examples

I.1	Example Client Task Program	I-1
I.2	Example Client Transparent Task-to-Task Program	I-6
I.3	Example Server Task Program	I-9
I.4	Example Client Task-to-Task Program	I-14

Tables

2-1	Remote Input File Transfers	2-5
2-2	Remote Output File Transfers	2-6
2-3	MS-DOS Functions	2-11
3-1	MS-DOS Function Requests for Transparent Intertask Communication	3-4
4-1	Socket Interface Calls	4-6
5-1	DECnet Utility Function Calls	5-5
6-1	NIOCB Data Fields	6-9
6-2	DECnet-DOS Network Process Calls	6-15
7-1	Command Line Switches for DNP with NETBIOS Emulation .	7-8
7-2	Network Control Block Fields	7-12
7-3	NETBIOS Emulator Commands	7-17
7-4	Digital Session Control Block (DSCB) Fields	7-20
7-5	DEC Extended Function Commands	7-21
8-1	CTERM Functions	8-6
8-2	CTERM Reason Codes	8-7
9-1	LAT Call-Back Routine Data	9-9
9-2	LAT Functions	9-12
D-1	Extended Error Messages – Unable to Make a Connection	D-1
D-2	Extended Error Messages - Disconnecting a Logical Link	D-4
G-1	DAP Maccode Field Values	G-2
G-2	DAP Miccode Values for Use with Maccode Values of 2, 10, 11	G-3
G-3	DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7	G-9
G-4	DAP Miccode Values for Use with Maccode Value 12	G-18

Figures

1-1	Personal Computers in an Ethernet LAN	1-5
1-2	Sockets Basic Building Blocks for Intertask Communication .	1-14
1-3	Establishing a DECnet Logical Link	1-17
1-4	Sending Data Messages over a DECnet Logical Link	1-20
1-5	Receiving Data Messages over a DECnet Logical Link	1-21
1-6	Closing Down the DECnet Logical Link Connection	1-23
6-1	An NIOCB Data Structure	6-8

1. The first part of the report is a general introduction to the project. It describes the purpose of the study and the objectives that were set at the beginning. It also mentions the scope of the work and the limitations of the study.

2. The second part of the report is a literature review. It discusses the work that has been done in the field of research and identifies the gaps in the knowledge. It also mentions the theoretical framework that was used in the study.

3. The third part of the report is a description of the methodology that was used in the study. It includes information about the data collection methods, the sample size, and the statistical analysis that was performed.

4. The fourth part of the report is a presentation of the results of the study. It includes tables, figures, and text that describe the findings of the research.

5. The fifth part of the report is a discussion of the results. It interprets the findings and discusses their implications for the field of research. It also mentions the limitations of the study and suggests areas for future research.

6. The sixth part of the report is a conclusion. It summarizes the main findings of the study and provides a final statement about the research.

Preface

DECnet-DOS is a communications software product that lets you use your IBM[®] personal computer in a DECnet[™] network. DECnet-DOS[™] Version 3.0 runs on supported IBM personal computer systems using the PC DOS operating system. For a list of all supported personal computers (such as IBM systems and industry-compatible personal computers) and a list of supported operating systems, refer to the software product description that came with your DECnet-DOS V3.0 kit.

DECnet-DOS provides several facilities for MS-DOS[®] application programmers to use in developing distributed programs. These facilities include several levels of task-to-task communications as well as file access and terminal access.

Manual Objectives

The *DECnet-DOS Programmer's Reference Manual* documents the various programming interfaces to the DECnet-DOS network software. In addition, this manual documents the DECnet-DOS language library and programming utilities that you can use to write DECnet-DOS application programs. This manual also discusses software considerations you should be aware of when developing DECnet-DOS applications in C or assembly programming languages.

Intended Audience

This manual is designed for programmers who are responsible for creating DECnet-DOS network applications. The discussion of the network software is restricted to the DECnet-DOS node. Therefore, this manual assumes the reader has experience programming for the MS-DOS or PC DOS operating system and is familiar with DECnet terms and concepts. For more information about DECnet terms and concepts, you should refer to the *DECnet-DOS User's Guide* or to the *DECnet-DOS Network Management Guide*. It is also helpful if you have experience writing DECnet applications.

DECnet-DOS Features

DECnet-DOS Version 3.0 provides several features for network application programming. The *DECnet-DOS Programmer's Reference Manual* provides information for the following features:

- NETBIOS emulation session interface
- Command Terminal (CTERM) interface
- Local Area Transport (LAT) interface
- Support for medium- and large-model programming
- Enhancements to the assembly language interface to the network process
- Additional programming examples

How to Use This Guide

This manual is divided into three parts:

Part I provides introductory material and general instructions for the inexperienced network programmer.

- **Chapter 1** presents an introduction to DECnet and the DECnet-DOS programming environment.
- **Chapter 2** describes how to use the DECnet-DOS Transparent File Access (TFA) programming utility to access files on remote nodes. The chapter describes how to use TFA in conjunction with your standard MS-DOS applications.
- **Chapter 3** describes how to use the DECnet-DOS Transparent Task-to-Task (TTT) programming utility to write applications that implement transparent network task-to-task communications.

Part II describes the various programming interfaces to the DECnet-DOS network software for the more experienced programmer.

- **Chapter 4** describes how to use C language function calls to the socket interface to write network applications that implement nontransparent task-to-task communication. This chapter provides detailed descriptions for each call (including syntax and diagnostic information) and provides instructions for linking your application with the DECnet routines found in your DECnet-DOS programming library.

- **Chapter 5** documents the C language programming interface library. This chapter describes the DECnet function calls you use to access the socket-interface library subroutines found in your DECnet-DOS programming library. You use these calls when writing nontransparent task-to-task network applications. This chapter also provides instructions for linking your application with the DECnet library routines.
- **Chapter 6** documents the assembly language interface to the DECnet-DOS network process (DNP). This chapter describes how to use DNP calls to write network applications that implement nontransparent task-to-task communications.
- **Chapter 7** documents the functional interface to the NETBIOS emulation interface. This chapter describes how to use the NETBIOS interface to write network applications that access network functions and communicate with other NETBIOS applications over the DECnet network.
- **Chapter 8** documents the functional interface to the Command Terminal (CTERM) software module. This chapter describes how to use the CTERM interface to write network applications that access the CTERM software and network services.
- **Chapter 9** documents the functional interface to the Local Area Transport (LAT) software module. This chapter describes how to use the LAT interface to write network applications that access the LAT software and network services.

Part III provides reference information in the following appendixes:

- **Appendix A** lists common definitions for the socket network interface.
- **Appendix B** details the format of the data structures used with the socket interface and assembly language network process interface calls.
- **Appendix C** summarizes the DECnet error completion codes.
- **Appendix D** summarizes the extended DECnet error codes.
- **Appendix E** describes how to use the TNT utility to obtain extended error messages for transparent file access and transparent task-to-task operations.
- **Appendix F** lists the extended messages you can receive for transparent file access operations.

- **Appendix G** lists Data Access Protocol (DAP) error messages that provide extended error information for transparent file access operations.
- **Appendix H** lists the specific socket interface and assembly language DECnet network process interface calls which cannot be transported to a DECnet-ULTRIX™ system.
- **Appendix I** contains C programming examples using socket interface function calls.

In this manual, the term “DOS” refers to any vendor-specific personal computer DOS operating system.

If you are unfamiliar with DECnet-DOS and/or DECnet task-to-task programming concepts, you may want to read through all of Part I before you start writing or modifying network applications.

If you are familiar with DECnet-DOS programming concepts and wish to write non-transparent task-to-task applications to access the DECnet-DOS socket interface, you may want to review Chapter 1 first, then refer to Chapter 4, 5, or 6 for detailed reference.

If you wish to write task-to-task applications that can communicate with NETBIOS applications over the DECnet network, you may want to read through Chapter 1 first and then refer to Chapter 7 for more information.

Associated Documents

The following documents are included in the DECnet-DOS documentation set:

- *DECnet-DOS Getting Started*

This manual provides an overview of DECnet-DOS. It includes an explanation of the product set, a road map through the documents, a brief description of DECnet-DOS utilities and the tasks that can be performed with them, and an introduction to frequently used DECnet-DOS commands.

- *Installing DECnet PCSA Client for DOS (with Diskettes)*

This manual describes procedures for installing and verifying the DECnet-DOS software on IBM personal computers and selected industry-compatible personal computers. This manual addresses a varied audience by providing the following:

- instructions for using the automated installation procedure (for the non-technical end user).
- reference material for the more technically oriented user who wishes to by-pass the automated system prompting.

- *DECnet-DOS User's Guide*

This manual describes the basic set of DECnet-DOS utilities and how to use them. It describes the DECnet-DOS commands and command syntax in detail.

The User's Guide describes these DECnet-DOS utilities:

- Network File Transfer (NFT)
- File Access Listener (FAL)
- Job Spawner (Spawner)
- Network Device Utility (NDU)
- DECnet-DOS Mail Utility (Mail)

- *DECnet-DOS SETHOST Terminal Emulation Guide*

This manual describes the SETHOST network virtual terminal utility. It describes how to use SETHOST and the set-up screens to connect to a host node and emulate a terminal connected to the host node.

This manual also describes how to use scripts, which are text files containing commands that allow SETHOST to perform many operations automatically.

- *DECnet-DOS Network Management Guide*

This manual describes the Network Control Program (NCP). The manual describes how to use NCP to manage your personal computer network, monitor the local node and remote nodes, and perform loopback tests for troubleshooting. The management guide also details the commands and command syntax for NCP.

You may also need to refer to the the following manuals:

- The *IBM Local Area Network Technical Reference*, Order number SC30-3383-2.
- The *IBM NETBIOS Application Development Guide*, Order number 68X2270.
- Any DOS Technical Reference manual.

Graphic Conventions

Convention	Meaning
bold	Words in boldface are considered to be literal and are typed exactly as shown. The call name is always shown as literal.
<i>special type</i>	This special type indicates system output or user input. System output is in black; user input is in red.
<i>italics</i>	<i>Italics</i> in commands and examples indicates a value that either the system supplies or you should supply.
CTRL/x	CTRL/x indicates that you should hold down the CONTROL key while you press the x key, where x represents a specific key.
key	Indicates that you should press the specified key.
<code>#include <file.h></code>	When <code>#include</code> appears in a syntax section, it indicates an include file or a header file. This type of file contains a collection of definitions commonly used throughout a program. You must include this file whenever it is required by a specific function call.

Part I

1763

Introduction to Network Programming

This chapter describes DECnet-DOSTM concepts and introduces the programming interfaces supplied with the DECnet-DOS product as follows:

- Sections 1.1 and 1.2 introduce DECnetTM and DECnet-DOS.
- Section 1.3 describes the programming interfaces supplied with the DECnet-DOS product.
- Section 1.4 reviews DECnet network programming concepts and considerations.

If you have not used DECnet-DOS or read about programming DECnet-DOS applications, you should read through this introductory chapter.

If you are familiar with basic DECnet concepts but unfamiliar with the DECnet-DOS implementation, you may want to begin your reading with Section 1.2, then read through the rest of the chapter.

If you have experience writing DECnet-DOS network applications, you may want to briefly review descriptions of the available interfaces in Section 1.3. In addition, you may find it helpful to review the network concepts presented in Section 1.4.

1.1 What Is DECnet?

DECnet is Digital's family of software and hardware communications products that enable users of various Digital computer systems to participate in a computer network. DECnet's peer-to-peer network environment allows any computer (or node) to communicate with every other node in the network without consulting a central controlling node. In this environment, each node is equally responsive to user requests, allowing network users to easily gain access to applications and facilities on other nodes. This simplifies communications and data handling, and provides flexibility when configuring a network. DECnet software, located on each system in a DECnet network, provides users with an interface that extends their computer's operating system. This network interface lets users communicate and share resources with other users in the DECnet network. Users in a network can exchange information, files, and programs.

Digital provides DECnet software for different operating systems, including the following:

- DECnet-VAXTM for VAXTM computers running the VMSTM operating system.
- DECnet-ULTRIXTM for VAX computers running the ULTRIXTM operating system.
- DECnet-DOS for IBM[®] personal computers running the PC DOS operating system. Refer to the *DECnet-DOS Software Product Description* (SPD) for a specific list of supported personal computers and operating systems.
- DECnet for OS/2TM, used on IBM personal computers and industry-compatible personal computers running the OS/2 operating system. Refer to the SPD for a list of supported personal computers and vendor-specific operating systems.

1.1.1 How DECnet Components Communicate

The Digital Network Architecture (DNA) is the set of rules, or protocols, that govern how DECnet software components communicate with each other. DNA is similar to the International Standards Organization's layered model for open systems interconnection. Its hierarchical structure supports communication between adjacent layers by using precisely-defined interfaces and protocols.

Because all DECnet components, or programs, implement DNA protocols, users can access other DECnet systems on the network regardless of system or operating system type. The DECnet software components, located on each DECnet node, handle the flow of information between nodes.

These software components intercept system-specific calls on a local node and translate them into requests for specific network services. DECnet components on a remote node then translate these calls into the same set of messages.

This program-to-program communication capability, called task-to-task communication, is the foundation for all other DECnet utilities. These utilities allow users to perform high-level network functions, including the following:

- **Remote file access** — Programs and users can access files that reside on remote nodes. Transport utilities permit the transfer of files between nodes. Users can create, store, and retrieve information on or copy files to and from remote nodes.
- **Network terminal communication** — From their PC or terminal, users can log on to a remote node and execute commands as if they were typing on a local terminal at that node.
- **Network management** — Management and maintenance utilities allow users to control, monitor, and test network activity.
- **Network resource sharing** — Virtual device utilities let users access remote disk drives and printers as if they were directly connected to the local computer.

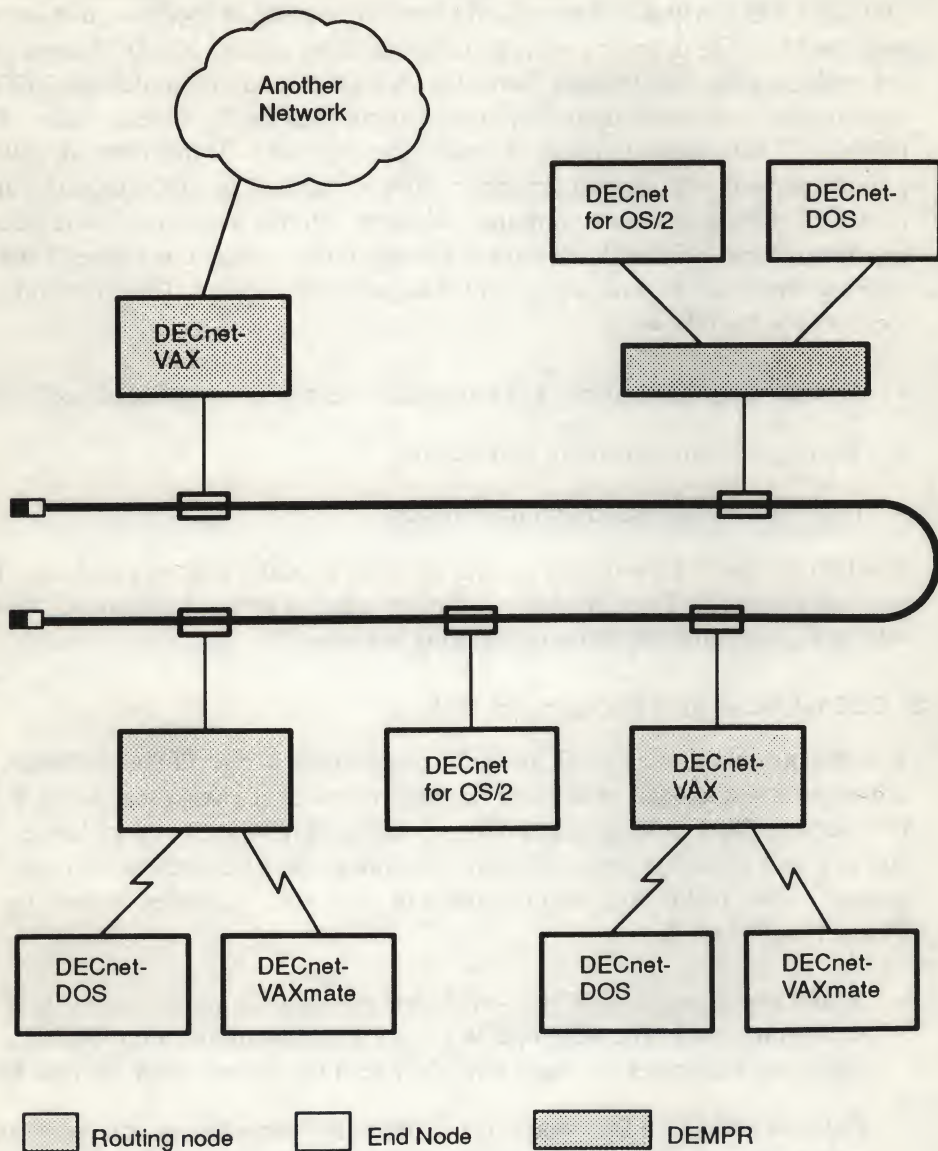
1.2 What Is DECnet-DOS?

DECnet-DOS is DECnet software that allows personal computers, running the MS-DOS[®] or PC DOS operating system, to participate as nonrouting nodes in a DECnet network. The DECnet-DOS software kit provides many separate components and utilities that let you do the following tasks:

- Exchange information with other users in the network.
- Copy, delete, or display files on remote nodes.
- Log on to remote systems.
- Monitor network activity.
- Share resources (such as disks and printers) with other users in the network.

Figure 1-1 shows personal computers running DECnet-DOS in an Ethernet local area network.

Figure 1-1: Personal Computers In an Ethernet LAN



LKG-3087-891

1.2.1 DECnet-DOS Components Work with Your Operating System

DECnet-DOS consists of a set of software components, or modules, that interface with the MS-DOS operating system. These modules exist as MS-DOS device drivers, resident tasks, and utilities. Some DECnet-DOS modules implement DECnet's task-to-task communications by intercepting MS-DOS system calls. These DECnet-DOS modules translate the intercepted system calls into network calls that provide network services to user applications. Other DECnet-DOS modules implement DECnet's task-to-task communications by offering a direct software interrupt interface. These DECnet-DOS modules accept interrupt function requests and provide user applications with access to DECnet network services. These network services include the following:

- Establishing, maintaining, and terminating network communications links.
- Transmitting and receiving information.
- Providing status and control information.

The DECnet-DOS kit provides several software modules that let you access these network resources. These modules comprise network kernel components, resident interfaces, user utilities, and programming libraries.

1.2.2 DECnet Runs As a Background Task

You need a minimum set of DECnet-DOS components to run DECnet software. This minimum configuration, or network "kernel," operates as a background task on your PC. Because DOS is not a multitasking operating system, the network kernel operates as a background process so that you can run another PC application in the foreground. The following memory-resident software modules make up the DECnet-DOS kernel.

- Real-Time Scheduler (SCH) — SCH lets the network kernel operate as a background task. The SCH operates as a communications interrupt shell, managing interrupts and regulating the use of the system clock on your PC.
- Datalink (DLL) — The DLL driver handles the transmission and reception of messages on the Ethernet. Device-specific DLL drivers work with the DNP and LAT modules in an Ethernet environment.

- **DECnet-DOS Network Process (DNP)** — DNP processes all requests for network services. The DNP provides all the network services for asynchronous configurations and works with DLL drivers to provide network services for Ethernet LAN configurations.

In addition to the kernel components, the following modules are often used for virtual terminal access:

- **Local Area Transport Process (LAT)** — The LAT module provides services, which are equivalent to those provided by Ethernet Terminal Servers, for network applications in an Ethernet environment. LAT services include support for multiple terminal sessions.
- **Command Terminal (CTERM)** — The CTERM module provides Wide Area Network (WAN) services to terminal emulators and other network applications for Ethernet or asynchronous DDCMP configurations. CTERM services include support for multiple terminal sessions.

1.3 Which Programming Interface Should You Use?

DECnet-DOS provides different levels of entry to the DECnet network process for network programmers. You can access these levels through any one of the following resident programming interfaces:

- **Transparent File Access**
- **Transparent Task-to-Task**
- **C Language Nontransparent Task-to-Task**
- **NETBIOS Emulation**
- **Assembly Language Nontransparent Task-to-Task**

In addition, DECnet-DOS provides a programming interface to the LAT and CTERM modules.

The interface you choose to use depends upon several factors, including:

- The type of task you want your program to complete.
- Whether you are modifying an existing program or creating a new program.
- The level of control over the communications process that you want your program to have. For example, do you need to be notified immediately of certain network events?
- The programming language you want to use.
- Your programming skill, knowledge of DOS, and knowledge of DECnet networking concepts.

DECnet-DOS offers programming interfaces that allow you to write task-to-task applications. Before you choose an interface, you should review the following definitions.

1.3.1 Task-to-Task Communications

Programmers include DECnet calls in cooperative programs to let the programs communicate with each other over a DECnet network. These calls activate DECnet routines that perform specific functions such as the creation and control of a communications session called a "logical link." This capability is called task-to-task communications.

Performing task-to-task communications is similar to performing input/output (I/O) operations. The logical link between the two programs is like an I/O channel over which both programs can send and receive data.

DECnet task-to-task calls in communicating programs perform several functions. They include:

- Requesting a logical link
- Receiving a logical-link request
- Accepting or rejecting a logical-link request

After a logical link is established, programs use task-to-task calls to do the following:

- Send or receive data
- Send or receive interrupt data
- Terminate the logical link

The number of calls you need to effect any of these depends on the programming language and programming interface you use. In some cases, a program may have to issue three separate calls to perform a network function; in other cases, a program may need to issue only one call.

1.3.1.1 Transparent and Nontransparent Task-to-Task

This manual uses the terms “transparent” and “nontransparent” when describing types of task-to-task communications and programming interfaces. Transparent refers to a function that you can use without seeing it. For example, when you issue a command to an operating system, the command interpreter parses the command string and invokes the appropriate system software. You see only the result of the processing, not the processing itself.

Similarly, a transparent programming interface provides functionality to the network using existing interfaces. You do not see the processing that occurs in the background between the interfaces.

A nontransparent programming interface not only lets you see the processing that goes on “behind” it, but requires that you understand the processing as well. Using a nontransparent interface for task-to-task applications offers you the greatest control over the network functions.

1.3.2 Resident Interfaces

The following sections provide more information about each programming interface. You should review these sections before you decide which interface best suits your programming needs.

1.3.2.1 Transparent File Access (TFA) and Transparent Task-to-Task (TTT)

The TFA and TTT modules provide network services to your simple DOS applications. When TFA is installed on your system, applications that use DOS interrupts to perform standard file I/O operations can perform the same operation to a sequential file on a remote system. With TTT, your PC application, written in any high-level language, can perform standard I/O operations and communicate with an application on another network node.

TFA and TTT modules intercept the standard DOS interrupts that your program uses to request I/O operation to another program. TFA or TTT examines the file specification associated with each interrupt to determine whether the request should be processed by the network or by the operating system. If the file specification is a standard DOS file specification, TTT or TFA passes the request to DOS for processing. If the file specification includes a special network string, TTT or TFA processes the request, substituting network services for the DOS processing.

For more information about TFA, refer to Chapter 2. For more information about TTT, refer to Chapter 3. These chapters explain what you need to know to modify your existing programs so that they can access network services.

1.3.2.2 C Language Task-to-Task Interface

This interface supports the DECnet-ULTRIX task-to-task network communications interface. Your C language applications can access all of the functions of the DECnet-DOS session layer through the C language task-to-task interface. This interface offers a broader range of network functions and control to your application program than offered by the TFA or TTT modules.

DECnet-DOS provides C language source and header files that you can use to create a linkable library for your network applications. Your program can issue simple calls to C language subroutines. These subroutines handle network functions for your program. For example, one routine handles all the steps involved in establishing a communications session. Chapter 4 describes how to issue calls to these DECnet subroutines.

If you need to have more control over the communications process, your program can issue calls to individual DECnet functions. To use these functions effectively, you should understand DECnet network concepts such as the steps involved in creating a communications link or manipulating your node database. Chapter 5 discusses how to issue DECnet function calls. However, before you start writing your network application, you should review the networking concepts presented in Section 1.4 of this chapter.

1.3.2.3 NETBIOS Emulation

The NETBIOS emulator is a Digital developed emulation of the industry-standard NETBIOS interface. NETBIOS is a session-level interface for network applications that are written to use the IBM PC-Network Adapter Card or the IBM Token Ring Network Adapter Card. With NETBIOS installed on your PC, your industry portable application can communicate with another NETBIOS application, or DECnet application, over a DECnet network.

Your application accesses NETBIOS services through software interrupt 5CH or 2AH. Your application issues function calls to open or close communication links and to send and receive data.

Chapter 7 describes the NETBIOS interface. Because the NETBIOS calls your application issues are mapped to calls to the DNP, you may also need to use Chapter 6 for a reference. However, before you start writing your application, you should review the networking concepts presented in Section 1.4 of this chapter.

1.3.2.4 Assembly Language Task-to-Task Interface

Your program can access all of the functions of the DECnet-DOS Session layer with the assembly language task-to-task interface. This interface offers your program the greatest level of control over network functions and, consequently, requires a thorough understanding of network programming.

Your application sends data to the DECnet-DOS Network Process through software interrupt to vector 6EH. Your application issues function calls to open or close communication links and to send and receive data.

Chapter 6 describes the assembly language task-to-task interface. However, before you start writing your application, you should review the networking concepts presented in Section 1.4 of this chapter.

1.3.2.5 Command Terminal (CTERM) Interface

The Command Terminal (CTERM) module provides a way for terminal emulators and other programs to communicate with a host system in a DECnet network. CTERM provides Wide Area Network (WAN) or Local Area Network (LAN) services to applications connected to a host through Ethernet or asynchronous DECnet connections.

Application programs, such as the DECnet-DOS SETHOST virtual terminal, can access CTERM services. These services include:

- Functions to manage a terminal session to a remote DECnet node.
- Support for multiple interactive terminal sessions.

Your application accesses CTERM services by issuing a software interrupt to vector 69H. Chapter 8 describes the programming interface to the CTERM module.

1.3.2.6 Local Area Transport (LAT) Interface

In LAN configurations, the LAT module provides network services to terminal emulators and other programs. Your program, such as a terminal emulator, can use LAT to communicate with a remote host system over a ThinWireTM or standard Ethernet network.

Your application accesses LAT functions by issuing software interrupts to vector 6AH. Chapter 9 describes the programming interface to the LAT module.

1.4 Network Programming Concepts and Considerations

This section provides an introduction to the networking concepts you need to know before you can create a DECnet-DOS network application. Review the concepts presented in this section before you turn to the chapters that describe specific interfaces.

1.4.1 The Client-Server Model

Before you can create a DECnet-DOS application, you need to understand the following programming concepts:

- **Client and server tasks.** Client and server tasks communicate through sockets. These tasks exchange data over logical links.

- **Sockets.** Sockets are the basic building blocks for DECnet-DOS task-to-task communication. They are created by tasks for sending and receiving data. They contain information about the status of the logical link connection.

1.4.1.1 Client and Server Tasks

DECnet-DOS communication requires cooperation between two programs or tasks. For the purposes of defining the DECnet-DOS programming interface, a distinction is made between the client task which initiates a connect request, and the server task which waits for and accepts or rejects the connection.

A client task is the program which initiates a connect request with another task. A server task is the program which waits for and accepts/rejects the pending connect request.

Once a logical link is established, the client and server tasks have a peer-to-peer relationship. The operations performed on their respective sockets are symmetrical. Either task can act as the source or receiving task and can send and receive data, or terminate the logical link at any time.

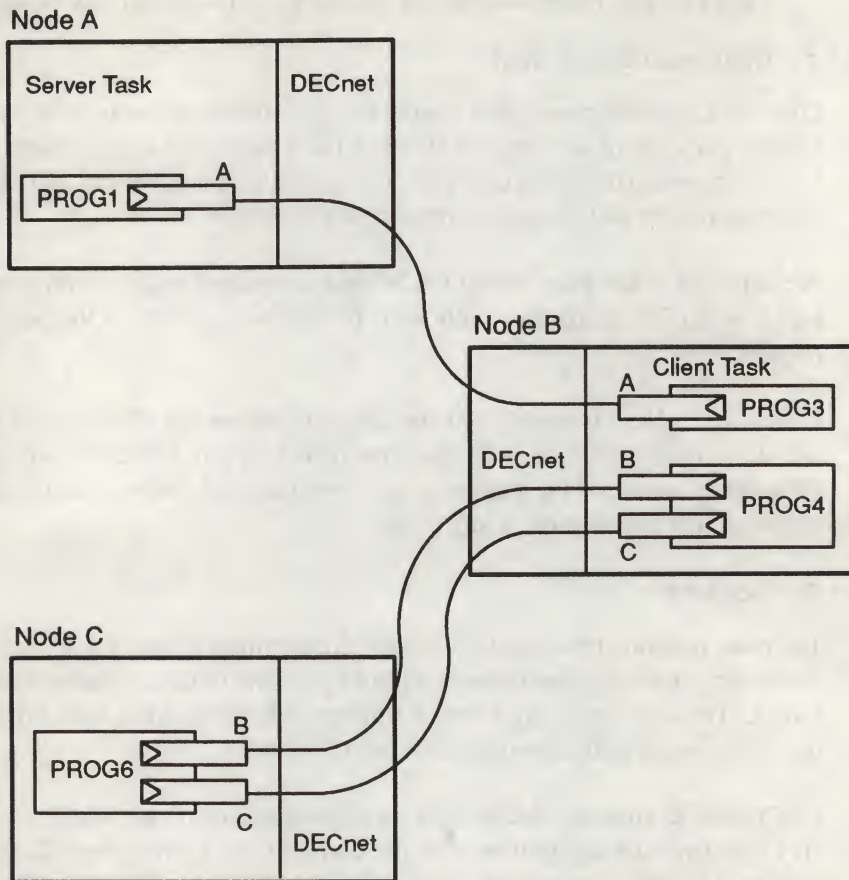
1.4.1.2 Sockets

The basic building block for DECnet-DOS communication is the socket: an addressable endpoint of communications within a program or task. A task uses the socket to send and receive data to and from a similar socket in another task. Figure 1-2 illustrates the use of sockets within DECnet-DOS tasks.



DECnet-DOS supports stream sockets and sequenced packet sockets. Stream sockets cause bytes to accumulate until internal DECnet buffers are full. The receiving task does not know how many bytes were sent in each write operation. Sequenced sockets cause bytes to be sent immediately. The receiving task receives those bytes in one "record."

A DECnet-DOS program can detect any potential problems by polling the socket's status, or by receiving error status in response to network requests.

Figure 1-2: Sockets: Basic Building Blocks for Intertask Communication



Legend:

-  Socket
-  Program
- A** Logical link between programs PROG1 and PROG3 (using sockets)
- B** Logical link between programs PROG6 and PROG4 (using sockets)
- C** Logical link between programs PROG6 and PROG4 (using sockets)

LKG-3081-89I

1.4.2 How Tasks Communicate

This section describes the functions that the client and server tasks request to communicate with each other. Illustrations that appear in this section use the socket interface calls to show task-to-task communication capabilities.

1.4.2.1 Establishing a Logical Link

The creation of a logical link is a cooperative venture. Two tasks must agree to communicate before you can have an established logical link. The process of establishing a logical link is detailed in Figure 1-3. A logical link connection is required before data can be exchanged between two tasks.

To begin the process, the server task creates a socket supported by DECnet. When this socket is first created, it has no assigned name or number. An object name or number is assigned to the socket. The name or number is required for use in future listening operations. The socket declares itself as a server which is available for client connections.

In turn, the client task must create a socket supported by DECnet. The client task can set up access control information and/or optional connect data. (See Sections 1.4.2.5 and 1.4.2.6 for an explanation of each.) The system returns an integer value called a socket number. Subsequent DECnet-DOS function calls on this socket will reference the associated socket number. At this point, the client task requests a logical link connection to another task. Any optional user data and/or access control information is sent along with the connection request.

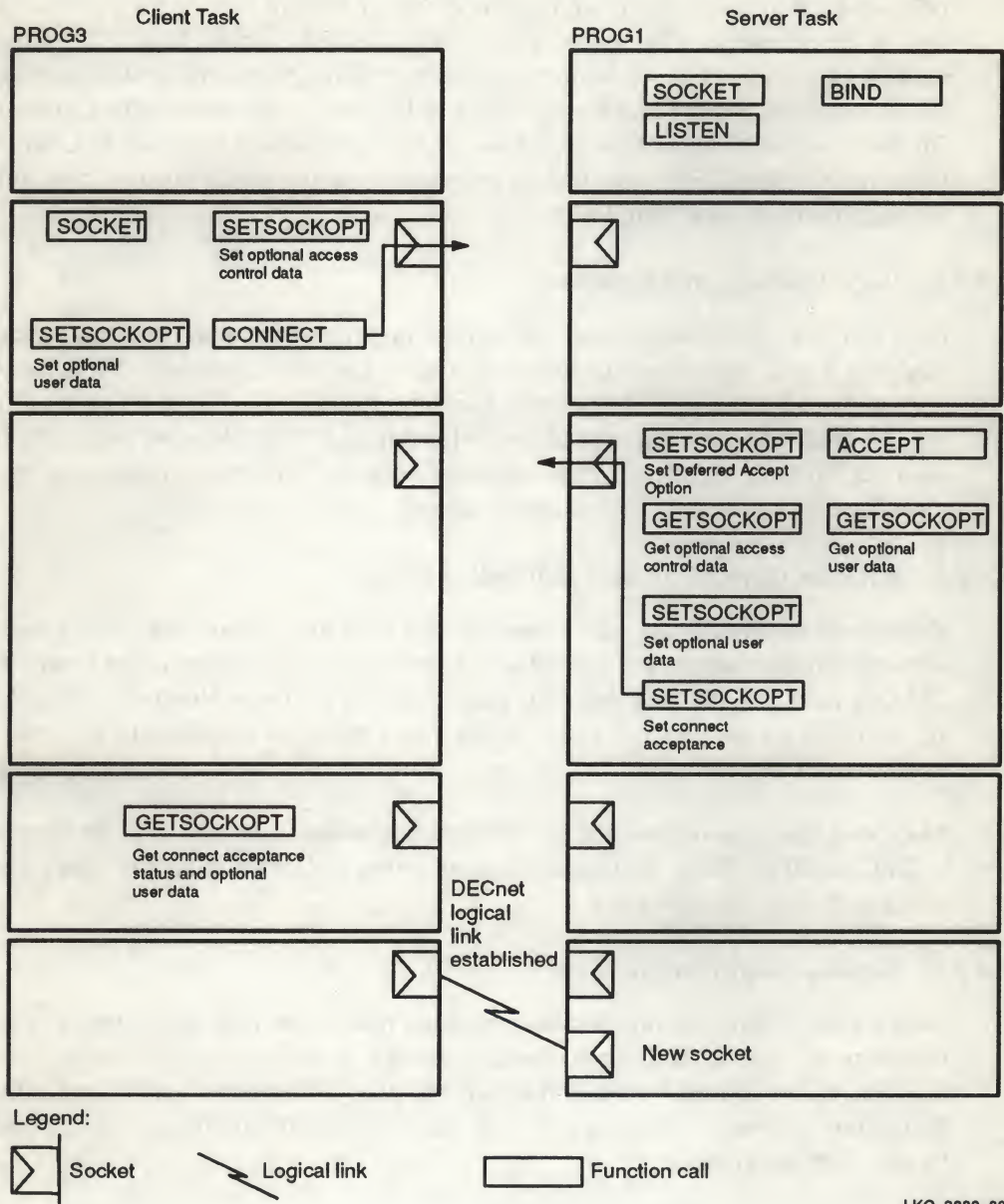
The server task can define how it accepts or rejects an incoming connection request. There are two options:

- The server task can immediately accept the connection request. In this case, none of the optional user data and/or optional access control information is used by the task. A logical link has successfully been established between the two tasks. Another socket is also created for the server task. Using the new socket, the server task can exchange data with the client task. The original socket remains open for the server task to listen for other incoming connection requests.

- For nontransparent communication only, the server task can defer making a decision about accepting or rejecting the incoming connection request. When the deferred option is set, the server task can examine any optional user data and/or optional access control information. It can then send optional user data with an acceptance or rejection message to the client task. The client task then retrieves the optional user data and/or status message.

If the connection fails, another attempt can be made at establishing a logical link. If the connection succeeds, the logical link is established and another socket is created for the server task. Using the new socket, the server task can exchange data with the client task. The original socket remains open for the server task to listen for other incoming connection requests.

Figure 1-3: Establishing a DECnet Logical Link



LKG-3082-891

1.4.2.2 Blocking and Nonblocking I/O Operations

DECnet-DOS allows tasks to send and receive data without waiting for the completion of the operation. This mode of operation is called "nonblocking I/O." When nonblocking is set, socket operations return to the calling program after the operation has been started, but not necessarily completed. Some operations must be restarted. On the other hand, the default mode for socket operations is blocking I/O. When blocking I/O is set, DECnet-DOS does not return control to the calling program until the operation has been completed.

1.4.2.3 Node Names and Addresses

Each system in a DECnet network has a unique node name and address. A node name can have 1 to 6 alphanumeric characters with at least one alphabetic character. A node address is a binary number which consists of an area number and a node number. An area consists of a group of interrelated nodes. Multiple areas are typically used only in large networks. When initiating a connection with a remote node, you must identify that node with a name or address.

1.4.2.4 Network Object Numbers and Task Names

Client tasks can specify the server task that they want to communicate with by using network object numbers and task names. Network object numbers range from 1 to 255. Numbers 1 to 127 are assigned to generic network servers. Numbers 128 to 255 are available for user-written tasks. When a user specifies a task name, the object number must be zero.

The server task must be installed as a network task on the remote node. In the context of DECnet-DOS, the server task declares its network object number and task name with the BIND function call.

1.4.2.5 Access Control Information

Access control information contains arguments that define your access rights at the remote node. It consists of three character strings: user ID, password, and account number. Access control verification is performed according to the conventions of the destination system. For some systems, the access information is the log-in data used by the client program.

1.4.2.6 Optional User Data

DECnet-DOS allows up to 16 bytes of optional user data to be exchanged between tasks when connecting to/disconnecting from logical links.

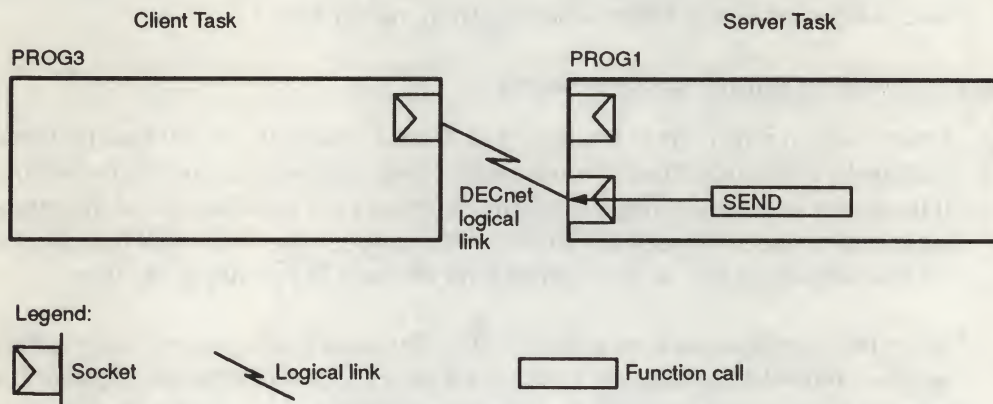
1.4.2.7 Sending Normal Data Messages

Either task can send normal data to its peer. It must specify the socket used for transmitting the message, the buffer containing the outgoing message, and the buffer size. If the socket is set to blocking I/O, and no buffer space is available to hold the outgoing message, the transmission is blocked until resources are freed up. If the socket is set to nonblocking I/O, an appropriate error message is returned to the user.

When the asynchronous form of the SEND call is used, control returns to the calling program immediately after the DECnet network process records the request. The network process may complete the request immediately or wait for a later time. To check to see if the data was sent, you can either use a callback routine or poll for status. Chapter 6 details the asynchronous form of the SEND call.

Figure 1-4 shows normal data being sent over a logical link.

Figure 1-4: Sending Data Messages over a DECnet Logical Link



LKG-3083-891

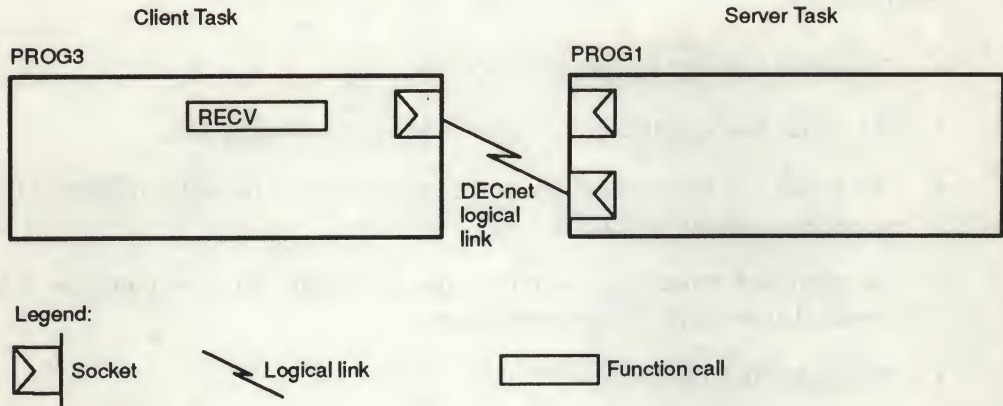
1.4.2.8 Receiving Normal Data Messages

Either task can receive normal data from its peer. It must specify the socket used for receiving the message, the address of the buffer which will store the incoming message, and the buffer size. If the socket is set to blocking I/O, and no messages are received, the task waits for a message to arrive. If the socket is set to nonblocking I/O, and no data is ready to be received, an appropriate error message is returned to the user.

When the asynchronous form of the RCVD call is used, control returns to the calling program immediately after the DECnet network process records the request. The network process may complete the request immediately or wait for a later time. To check to see if the data was received, you can either use a callback routine or poll for status. Chapter 6 details the asynchronous form of the RCVD call.

Figure 1-5 shows data being received over a logical link.

Figure 1–5: Receiving Data Messages over a DECnet Logical Link



LKG-3084-891

1.4.2.9 Out-of-Band Messages

Out-of-band messages are unsolicited, high priority messages sent between non-transparent communication tasks over a logical link. An out-of-band message usually informs the receiving task of an unusual or abnormal event in the sending task. The valid range for the message size is 1 to 16 bytes.

Out-of-band messages are always sent ahead of outstanding normal messages. Unless certain error conditions exist, an out-of-band message is always sent even if the socket is set to blocking or nonblocking I/O.

1.4.2.10 Terminating Network Activity and Closing the Logical Link

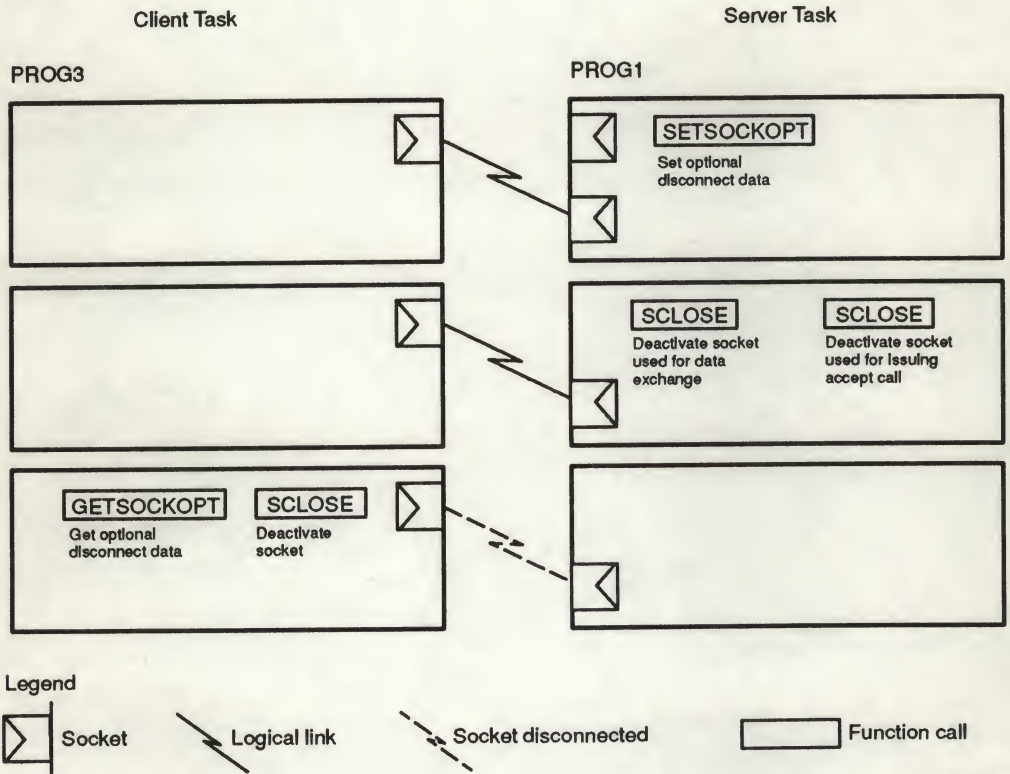
The process of terminating network activity can begin with either the client or server task. To initiate close-down for nontransparent communication, either task can send up to 16 bytes of optional disconnect data to the other task. The optional disconnect data is sent with an abort or disconnect option.

Figure 1-6 provides an example of how sockets are deactivated and how the logical link connection is broken. The close-down steps, as depicted in Figure 1-6, include the following:

- The server task sets up optional disconnect data.
- The server task deactivates its socket used for exchanging data.
- The server task can also deactivate the original socket on which it listened for incoming connection requests.
- The client task retrieves any optional disconnect data. At some point, the task should also deactivate its original socket.
- The logical link connection is broken.

Afterwards, the sockets are reclaimed as network resources that are made available for future assignment to this or another task.

Figure 1-6: Closing Down the DECnet Logical Link Connection



LKG-3085-891

THE UNIVERSITY OF CHICAGO

1961

1	2
3	4
5	6
7	8
9	10
11	12
13	14
15	16
17	18
19	20
21	22
23	24
25	26
27	28
29	30
31	32
33	34
35	36
37	38
39	40
41	42
43	44
45	46
47	48
49	50
51	52
53	54
55	56
57	58
59	60
61	62
63	64
65	66
67	68
69	70
71	72
73	74
75	76
77	78
79	80
81	82
83	84
85	86
87	88
89	90
91	92
93	94
95	96
97	98
99	100

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

THE UNIVERSITY OF CHICAGO

Transparent File Access (TFA) Interface

The DECnet-DOSTM Transparent File Access (TFA) module provides network services to your simple DOS applications. With TFA, your DOS application can access a file on another network node.

This chapter describes how to modify your existing programs that perform standard DOS file I/O requests so that they can access network services through the TFA interface.

2.1 Introduction

The TFA software module provides a means by which PC applications can access a file on another DECnetTM node. When TFA is installed on your system, applications that use DOS interrupts to perform standard file I/O operations can perform the same operations on a sequential file on a remote network node. For example, with TFA installed on your system, your application can:

- Create or open a remote file.
- Read from or write to a remote file.
- Load and execute a remote file.
- Search a remote directory.
- Close or delete a remote file.

The TFA software intercepts the standard DOS interrupt, INT 21H, which your program uses to request file I/O operations. TFA examines the file specification that is associated with each function request to determine whether the request should be processed by the network or by the operating system. If the file specification is a standard DOS specification, TFA passes the request to DOS for processing. If the file specification includes a special network string, TFA processes the request, substituting network services for the DOS processing. Refer to your DOS reference guide for a description of a valid DOS file specification.

2.2 TFA Command Line

DECnet-DOS provides TFA as an MS-DOS[®] terminate and stay resident task, TFA.EXE. You can invoke TFA by typing TFA at the DOS prompt or by adding the TFA command to a batch file, such as AUTOEXEC.BAT. Be sure to define a path to TFA.EXE, or ensure that it resides in your current default directory.

DECnet-DOS provides the Transparent Network Task (TNT) utility, which you can use to check for TFA errors. You can also use TNT to remove TFA from memory. Refer to Appendix E for instructions on using the TNT utility.

2.3 Accessing Remote Files

Your application uses DOS function calls and associated network file specification strings to send network I/O requests to TFA. This network string includes the name of the remote node, any user or password information required by the remote node, and the name of the file you wish to access.

The TFA software intercepts DOS interrupt 21H function requests and examines the file specification associated with each function request. If the file specification includes a special network string, TFA processes the request and substitutes network services for the DOS processing.

DECnet software handles all of the network access for your program. TFA routines build, send, receive, and interpret DECnet file access messages. These messages are defined by DECnet's Data Access Protocol (DAP), the protocol that governs remote file access over the DECnet network.

DECnet software, installed at the remote node, receives your request to access a file. This software, the File Access Listener (FAL), completes the connection request initiated by your application, translates your request to a system-specific call, then sends the request to the file system on the remote node. The remote FAL then sends the data back to the TFA module on your PC node. TFA interprets this data and forwards it to your application.

2.4 File Characteristics

A file has specific characteristics that determine how the file is transferred between local and remote systems.

The MS-DOS file system treats all files as a stream of 8-bit binary bytes. When accessing files or devices, you may use an ASCII or BINARY mode. In ASCII mode, read calls end with a carriage return character and the file ends with a Control-Z character. In BINARY mode, the data is read and written as the length of the requests.

On other file systems supported by DECnet, additional information is stored on the disk that allows access to the file's data, in units called "records." For example, on VMSTTM, the Record Management System (RMS) provides several forms of record organizations (such as sequential, relative, and ISAM); record formats (such as fixed, variable, variable with fixed header, and stream); and several different file access methods (such as sequential, random by record number, or random by string key).

Since MS-DOS cannot store this format or access information in a manner that would be transparent to the user of the data on the MS-DOS system, TFA presents the data in a format that is consistent with the data stream that would be seen by unit record equipment on that system.

The following section describes how TFA handles the file transfers and record attributes. Two concepts are used throughout, that of image as opposed to ASCII text files and that of stream as opposed to record file systems.

Declaring a file to be ASCII text allows TFA to handle the data in a way that record systems typically use to store text files, possibly interpreting text lines into records. On the other hand, the data in a file declared to be image will not be modified in any way.

2.4.1 File Attributes

File attributes for remote input files are determined by the remote file and the remote file system. For example, an ASCII file cannot be redefined as an image file.

File attributes for remote output files are determined by the format of the local file and the type of remote file system.

The set of file attributes is:

- **File Organization.** DECnet-DOS supports only sequential files. A sequential file has records arranged one after the other. The first record written is the first record in the file, and the record written most recently is the last record in the file.
- **Data Type.** A remote file can have an ASCII or image data type. Depending on the file's record attribute, record format and the remote file system type, DECnet-DOS can reformat ASCII data. On the other hand, DECnet-DOS cannot convert image data type files.
- **Record Format.** This particular file attribute indicates how records are formatted within the file. A record can have one of the following formats:
 - **Undefined** records have no declared formats.
 - **Stream** records consist of a continuous series of ASCII characters delimited by carriage return/line feed (CR/LF) pairs.
 - **Fixed length** records are identical in size.
 - **Variable length** records can be different lengths, up to a maximum size that you specify. The maximum size is fixed at file-creation time and cannot be changed for the life of the file.
 - **Variable with fixed length control (VFC)** records include a fixed-length control field that precedes the variable length data portion. This format allows you to construct records with additional data that label the contents of the variable length portion of the record.

- **Record Attributes (RATs).** This characteristic indicates how information is formatted within a record. Record attributes include implied carriage return/line feed (CR/LF) pairs, embedded carriage control, FORTRAN carriage control, print file carriage control, line sequence ASCII, and MACY11.
- **Fixed Header Size.** If the record format of the remote input file is VFC, this characteristic defines the size of each fixed length header.
- **Maximum Record Size (MRS).** If the record format is fixed, maximum record size defines the length of each record. For fixed length records, the default size is 128 bytes. For variable length records, MRS declares the maximum record size. There is no default size.

2.5 Performing Data Conversions

To read or write files on other nodes successfully, data must be formatted according to the conventions of the respective file system. A system running MS-DOS supports stream formatted files. Remote DECnet hosts can support stream and nonstream file systems: variable length, fixed length and VFC records.

The following tables summarize file structure interdependencies when you read or write files on remote systems.

Table 2-1: Remote Input File Transfers

File Type	Remote System Type	Record Attributes	Processing Mode
Image	Not applicable	Ignored	No conversion
ASCII	Stream	Ignored	No conversion with embedded carriage control
ASCII	Nonstream	Implied CR/LF pair	CR/LF pair appended to each record
ASCII	Nonstream	FORTRAN, Print or LSA carriage control	Conversion of carriage control done
ASCII	Nonstream	Null, embedded, block, MACY11	No conversion

Table 2-2: Remote Output File Transfers

File Type	Remote System Type	Record Attributes	Processing Mode
ASCII	Stream	None	No conversion. Records determined by LFs.
ASCII	Nonstream	Variable, implied CR/LF	New line and CR characters are dropped. Records determined by LFs.
Image	Stream	None	No conversion.
Image	Nonstream	Fixed: record size = 128 bytes.	No conversion. Last record is padded if necessary.

The following sections detail how TFA handles data conversions for remote input and output files.

2.6 Converting Remote Input Files

During remote file input, a file located on a remote DECnet host is transferred to an MS-DOS system. Local handling of the remote file is determined by its attributes and the type of remote file system.

2.6.1 Binary Image Files

No record to stream conversion is performed on a binary image file. The file is passed to the calling task one record at a time. The size of each record is set by the record attributes and the maximum record size of the remote input file. If the remote file system does not support record attributes (for example, stream), the size is set to 128 bytes.

The remote file's record format (RFM) and record attributes (RATs) are ignored in this type of file transfer.

2.6.2 ASCII Files

No data interpretation is performed on an ASCII file coming from a stream file system. For ASCII files being copied from a nonstream file system, data handling is determined by the remote file system, the remote file's record format and record attributes.

A nonstream file system can support variable length, fixed length, variable with fixed carriage control (VFC) record formats and stream record formats. It also supports a number of record attributes. The TFARs perform data conversion on ASCII files depending on the record attributes of the remote file. The following section describes how TFARs interpret data based on the remote file's record attributes:

- **RAT = Implied CR/LF.** No conversion of embedded carriage control characters. A CR/LF pair is appended to each record.
- **RAT = FTN, PRN.** FORTRAN (FTN) and print (PRN) carriage control characters are converted appropriately to stream file systems.
- **RAT = Null, embedded carriage control, block, MACY11.** No data is inserted between records.

2.7 Converting Remote Output Files

Remote output files are transferred from the local system to a remote DECnet host. By default, files are transferred in ASCII mode. The file's record format and record attributes are determined by the remote file system. A logical record consists of data up to and including a CR/LF pair. The TFARs send a file as image if the first logical record is not terminated by either a CR/LF or an LF/CR pair.

2.7.1 ASCII Files

If the local file is transferred to a remote stream file system, the logical records are passed with no data conversion being performed. The file structure defaults are:

Record Format	Record Attribute
Stream	NONE

If the file is copied to a nonstream file system, delimiting CR/LF pairs are dropped from the logical records before they are sent. The file structure defaults are:

Record Format	Record Attribute
Variable length	Implied CR/LF

2.7.2 Image Files

If the remote output file is an image file, the records are passed with the following default file structure:

Record Format	Record Attribute	Maximum Record Size (Bytes)
Fixed length	NONE	128

Since the default data type for a remote output file is ASCII, the TFARs initially create an ASCII file on the remote node. If the first logical record is not terminated by a CR/LF or an LF/CR pair, the TFARs will issue a DAP access message complete (purge) message. In this case, TFA creates a new image file on the remote node. This particular file is then sent using the data handling defaults for an image file transfer.

2.8 Accessing TFA Services

To access network services, your application must issue DOS INT 21H function calls, then pass a specially formatted network file specification string to TFA. TFA routines intercept these specific file I/O function requests and initiate the network processing that your application requests.

2.8.1 Network File Specification Syntax

The network file specification string you specify must include the format name of the remote node, any user or account information required by the remote node, and the name of the remote file you want to access.

You supply the elements for this string as follows (each element separated by a single backslash character):

\\node\access-control\filespec

where

- * double backslash indicates that this is a file specification associated with a network request.
- f* or *F* specifies that a data file will be accessed for this network operation. May be uppercase or lowercase.
- node* specifies either the name or address of the remote node. A node name has a maximum of 6 alphanumeric characters with at least one alphabetic character. A node address is a numeric string including the area number in the range of 1 to 63, and the node number in the range of 1 to 1023.
- access-control* provides user information required by the remote node. (You can also use NCP to set up access control information). You may supply data for the following elements:
- *user-id*
identifies a user name or log-in ID on the remote system. The user name and password set the user's privileges for accessing the remote task. A user name has a maximum of 39 characters.
 - *password*
defines a user's password that is associated with a *user*. A user's password has a maximum of 39 characters.
 - *account*
identifies a billing account number which is used with the user name and password information on some systems. An account number has a maximum of 39 characters. If the account information is not required, you can omit it from the string.

NOTE

If you do not supply access control information for the user ID, password, and account qualifiers, TFA will use the default access control information you set with the DECnet-DOS Network Control Program (NCP) command `DEFINE NODE`. If you have not defined access with NCP, TFA will attempt to process the request using only the node name you supply.

`\\` delimits the node specification from the file specification.

filespec specifies the name of the remote file you want to access. The file specification may include the device name and the directory in which the file is located, in addition to the file name, file extension, and version number.

File names must conform to the conventions of the target node. Any unspecified elements default to the target system's conventions. Refer to the remote system's documentation if you are not familiar with that system's file conventions.

2.9 Issuing Function Requests

Your application uses DOS function calls and associated network file specification strings to send network I/O requests to TFA as previously described.

Table 2-3 summarizes the MS-DOS function requests you use to invoke TFA network services.

Table 2-3: MS-DOS Functions

Hexadecimal Value	Function	Network Access
3CH	Create a file.	Initiate a logical link request to create a remote file.
3DH	Open a file.	Initiate a logical link request to open a remote file.
3EH	Close a file handle.	Close a remote file and terminate a logical link connection.
3FH	Read from a file/device.	Read data from a remote file.
40H	Write to a file/device.	Write data to a remote file.
41H	Delete a file from a specified directory.	Delete a file from a remote directory.
4BH	Load and execute a program.	Submit a remote command file to be executed.
4EH	Find first matching file.	Search for the first remote file that matches the specified file characteristics.
4FH	Find next matching file.	Find the next file entry that matches the name specified on the previous find first call.

2.10 TFA Programming Considerations

Your application accesses TFA services through DOS software interrupt 21H. Your application loads data into one or more processor registers. The contents of each register are defined by the specific MS-DOS function call. For each call, the AH register contains the function request code. Note that for all calls, the contents of the registers are preserved, except for those registers used to return results. For all calls, the carry flag is either cleared to indicate success, or set to indicate failure. If the request fails, more specific error codes are returned in the AX register.

When creating or modifying applications that access TFA, you should note the following:

- Some user programs may not accept the TFA network specification string.
- DECnet-DOS supports only MS-DOS V2.0 function calls. You should not use unsupported MS-DOS function requests to perform transparent file access operations.
- If you press **CTRL/C** while TFA is active, network operation may be blocked. To clear this condition, run the TNT utility.

Refer to the following section for detailed call descriptions.

3EH: Close a File Handle

Close – close a remote file, terminate a logical link connection, and deactivate the handle used for data exchange.

On Entry:

AH = 3EH

BX = handle for logical link

On Return:

If function succeeded:

Carry bit = clear

If function failed:

Carry bit = set

AX = error code

6 – Invalid handle value detected (specific to TFA)

DESCRIPTION

The *Close* function closes the remote file, terminates the logical link connection, and deactivates the handle used for data exchange.

On entry, the BX register contains the 16-bit handle value returned by the open or create I/O operation. If the close operation completes successfully, the carry bit is clear. If an error condition occurs, the carry bit is set and the appropriate error code is returned in the AX register.

3EH: Close a File Handle

NOTE

If you issue the *Close* call with a handle equal to -1, the TFARs will interpret the call as a request to abort any active Find Matching File operations.

3CH: Create a File

Create – initiate a logical link request to create a remote file.

On Entry:

AH = 3CH

DS:DX = address of remote file specification string

On Return:

If function succeeded:

Carry bit = clear

AX = handle for logical link

If function failed:

Carry bit = set

AX = error code

2 – Network process may not be loaded. Node name to node address mapping not found in database file. Target task on the outgoing connection not available. Network is unreachable.

3 – Target task not found.

4 – Too many active logical link connections.

5 – Remote object rejected the request.

3CH: Create a File

DESCRIPTION

The *Create* function call enables a source task to initiate a logical link request to create a remote file. When the request is made, the file is opened for write operations. On entry, DS:DX contains the address of the remote file specification string. Any optional access control information is passed as part of the string to the target task.

On return, the AX register contains an error code or a 16-bit handle associated with the source task. The returned handle value must be used for subsequent read and write I/O operations.

The TFARs exchange a series of DAP messages with the remote FAL in order to initialize the DAP environment and define the requested network access. Each link initialization involves an exchange of DAP configuration, attributes, and access messages. The configuration messages include information regarding the operating and file systems of the source and target systems, and the buffer size. The attributes messages supply information about the file to be accessed. Undefined file attributes are set to default values determined by the remote file system. The access message establishes the type of access to the remote file.

If you are unable to initiate a logical link connection, an error code is returned in the AX register.

41H: Delete a File

Delete – delete a file from a specified remote directory.

On Entry:

AH = 41H

DS:DX = address of remote file specification string

On Return:

If function succeeded:

Carry bit = clear

If function failed:

Carry bit = set

AX = error code

2 – Network process may not be loaded. Node name to node address mapping not found in database file. Target task on the outgoing connection not available. Network is unreachable.

5 – Remote object rejected the request.

DESCRIPTION

The *Delete* function call deletes a remote file from a specified directory.

On entry, DS:DX contains the address of the remote file specification string. Any optional access control information is passed as part of the string to the target task.

If an error condition occurs, the carry bit is set and the error code is returned in the AX register.

4EH: Find First Matching File

4EH: Find First Matching File

Find First Matching File – search specified remote directory for the first file that matches the specified file characteristics.

On Entry:

AH = 4EH

DS:DX = address of remote directory specification string

On Return:

If function succeeded:

Carry bit = clear

DTA = file name bytes 30-42
creation time bytes 22-23
creation data bytes 24-25
file size bytes 26-27 (low) bytes 28-29 (high)

If function failed:

Carry bit = set

AX = error code

2 – Network process may not be loaded. Node name to node address mapping not found in database file. Target task on the outgoing connection not available. Network is unreachable.

18 – There are no more matching files.

4EH: Find First Matching File

DESCRIPTION

The *Find First Matching File* function searches for the first file that matches the directory specification set by the user. If a directory specification is given without a file specification, or includes wildcards, the first matching file is returned. On entry, the DS:DX register contains the address of the network file specification string.

If a file is found that matches the specification string, the carry bit is cleared and the information is returned into the current Disk Transfer Data Block (DTA). The DTA is a portion of memory that is allocated for returning file data. The user can obtain a pointer to this area by issuing an MS-DOS software interrupt 21H *Get Disk Transfer Address* function 2FH.

If only one matching file is found, the carry bit is not set and an 18 (no more matching files) is returned in the AX register. If no matching file is found, the carry bit is set, and an error code of 2 (file not found) is returned in the AX register.

4FH: Find Next Matching File

4FH: Find Next Matching File

Find Next Matching File – find the next file entry using the context returned in the previous Find First call.

On Entry:

AH = 4FH

DTA = pointer to data block returned by function 4EH (find first matching file)

On Return:

If function succeeded:

Carry bit = clear

DTA = file name bytes 30-42
creation time bytes 22-23
creation data bytes 24-25
file size bytes 26-27 (low) bytes 28-29 (high)

If function failed:

Carry bit = set

AX = error code

18 – No matching files.

DESCRIPTION

This function call finds the next matching entry in a directory. On entry, the current DTA address must point to a data block returned by the previous *Find First Matching File* function call.

4FH: Find Next Matching File

If a file is found that matches the specification string, the information is returned into the current DTA data block. The DTA is a portion of memory that is allocated for returning file data. The user can obtain a pointer to this area by issuing an MS-DOS software interrupt 21H *Get Disk Transfer Address* function 2FH.

When a matching file is found, and it is the last matching file in the directory, an error code 18 is returned in the AX register and the carry bit is cleared. Likewise, if no matching file is found, the carry bit is set and an error code 18 (no matching file) is returned in the AX register.

4BH: Load and Execute a Program

4BH: Load and Execute a Program

Load and Execute a Program – submit a remote command file.

On Entry:

AH = 4BH

DS:DX = address of remote file specification string

ES:BX = address of parameter block, ignored

On Return:

If function succeeded:

Carry bit = clear

If function failed:

Carry bit = set

AX = error code

2 – Network process may not be loaded. Node name to node address mapping not found in database file. Target task on the outgoing connection not available. Network is unreachable.

4BH: Load and Execute a Program

DESCRIPTION

This function call allows an existing remote file to be submitted as a batch or command file for remote execution. The remote file is not deleted after completion of this call. On entry, the DS:DX register contains the address of the network file specification string. It specifies the remote file to be loaded and executed in a standard MS-DOS *Load and Execute* function call, and registers ES:BX points to a parameter block defining the command file's environment. These registers are ignored for remote command file submission.

If the load and execute is unsuccessful, the carry bit is set and the error reason is returned in the AX register.

3DH: Open a File

3DH: Open a File

Open – initiate a logical link request to open a remote file.

On Entry:

AH = 3DH

AL = access mode bits. Contains one of the following:

- 0 – open file for reading
- 1 – open file for writing
- 2 – open for reading and writing

DS:DX = address of remote file specification string

On Return:

If function succeeded:

Carry bit = clear

AX = handle for logical link

If function failed:

Carry bit = set

AX = error code

- 3 – Target task not found.
- 4 – Too many active logical link connections.
- 5 – Remote object rejected the request.

3DH: Open a File

DESCRIPTION

The *Open* call enables a task to initiate a logical link request to open a remote file. On entry, DS:DX contains the address of the remote file specification string. If you include wildcards as part of the specification string, only one file is opened.

Any optional access control information is passed as part of the string to the target task.

The access mode is defined in AL.

If the *Open* call completes successfully, a 16-bit handle is returned in the AX register. The handle value must be used for subsequent read and write I/O operations.

If an error condition occurs, the carry bit is set and an error code is returned in the AX register.

3FH: Read From a File/Device

3FH: Read From a File/Device

Read – receive data from a remote file.

On Entry:

AH = 3FH

BX = handle for logical link

DS:DX = address of file data buffer

CX = size of file data buffer

On Return:

If function succeeded:

Carry bit = clear

AX = number of bytes received from file

If function failed:

Carry bit = set

AX = error code

5 – Remote object rejected the request.

6 – Invalid handle detected.

3FH:Read From a File/Device

DESCRIPTION

The *Read* function call allows the target task to read data from a remote file.

On entry, the BX register contains the 16-bit handle value returned by the *Open* or *Create* call. The CX register contains the number of bytes to be received. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully received by the target task. If the carry bit is clear and AX=0, an end-of-file status is indicated. A single read function returns a maximum of one logical record.

If the buffer is too small for one logical record, no error occurs.

The next read continues to return bytes until the entire logical record has been read.

If an error condition occurs, the carry bit is set, and an error code is returned in the AX register.

40H: Write to a Remote File

40H: Write to a Remote File

Write – write data to a remote file.

On Entry:

AH = 40H

BX = handle for logical link

CX = size of file data buffer

DS:DX = address of file data buffer

On Return:

If function succeeded:

Carry bit = clear

AX = number of bytes written to file

If function failed:

Carry bit = set

AX = error code

5 – Remote object rejected the request.

6 – Invalid handle detected.

40H: Write to a Remote File

DESCRIPTION

The *Write* function call allows the source task to write data to a remote file.

On entry, the BX register contains the 16-bit handle value returned on the *Open* or *Create* call. The CX register contains the number of bytes to be sent. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully sent by the source task.

If an error condition occurs, the carry bit is set and an error code is returned in the AX register.

1000

1000

1000

1000

1000

1000

1000

1000

Transparent Task-to-Task (TTT) Programming Interface

DECnet-DOSTM supports transparent task-to-task communication for high level language and assembly language programs. DECnet-DOS supports the DOS Version 2.0 calls described in this chapter. Using specific calls, a task can perform standard I/O operations and can communicate with another task over the network.

3.1 Transparent Task-to-Task Communication

Transparent communication provides the basic functions necessary for tasks to communicate over the network. These functions include the initiation, acceptance, and establishment of a logical link; the orderly exchange of messages between DECnetTM tasks; and the controlled termination of the communication process.

When accessing the network transparently, you use no DECnet-specific calls to perform these functions. Instead, you use normal I/O statements provided by the applicable high level language. An assembly language task uses a subset of the MS-DOS[®] function requests to perform the same communication activities.

3.2 Transparent Communication Functions

This section describes the functions that the client and server tasks use to communicate over the network.

3.2.1 Initiating a Logical Link Connection

Transparent communication can take place only after a logical link is established between two cooperating tasks. You establish the logical link by issuing a client task call that requests a logical link connection. The request is sent to the server task on the remote node.

The interaction that takes place prior to establishing a logical link is termed a "handshaking sequence." Using transparent task-to-task communication, an MS-DOS program can act as either a client or a server.

3.2.2 Handshaking Sequence for a Client Task

To initiate the logical link request transparently, a client task performs a file create or open operation. This task supplies the following information:

- **The identification of the server node.** Every node in the network has a unique identifier that distinguishes it from other nodes in the network. Transparent communication uses a node specification string to indicate the name of the server node.
- **The identification of the server task.** Client tasks specify the server task that they want to communicate with by using a network task specification string. This string uses network object numbers and task names. Network object numbers range from 1 to 255. Numbers 1 to 127 are reserved for Digital use. Numbers 128 to 255 are available for user-written tasks.

When a user specifies a task name, the object number is zero.

High level language client tasks can use standard file opening statements to request a logical link connection to the remote task. An assembly language client task uses the MS-DOS *Create* or *Open* function request to perform the same operation.

3.2.3 Handshaking Sequence for a Server Task

A high level language server task performs a file create or open operation to accept the logical link connection request. If *SYS\$NET* is specified as the node name, the task is always a server task. An assembly language server task can accept the logical link request with either the MS-DOS *Create* or *Open* function request.

3.2.4 Exchanging Data Messages over a Logical Link

Once the logical link is established, either task can send and receive data messages. A coordinated set of write and read operations enables the exchange of data over the logical link. For high level language tasks, standard read and write calls are used for data exchange. An assembly language task uses the MS-DOS *Read* and *Write* function requests. The handle returned by the previous *Create* and *Open* function requests must be specified in all *Read* and *Write* function requests.

3.2.5 Terminating the Logical Link

The termination of a logical link signals the end of the communication process between two tasks. When network activity is no longer required, either high level language task can issue a file closing statement to break the link. Likewise, either assembly language task can issue the MS-DOS *Close* function request to terminate the connection. This particular MS-DOS call closes the logical link and deactivates the original handle used for data exchange.

3.3 Command Line for Transparent Communication Tasks

DECnet-DOS provides the Transparent Task-to-Task (TTT) utility as an MS-DOS terminate-and-stay-resident task, TTT.EXE. You can invoke TTT by typing TTT at the DOS prompt. For example:

```
E>TTT Return
```

The system responds with either:

```
DECnet - TTT Version 3.0 installed
```

or

```
DECnet - TTT Version 3.0 has already been installed
```

You can also invoke TTT by adding the TTT command to a batch file, such as AUTOEXEC.BAT. Be sure to define a path to TTT.EXE, or ensure that it resides in your current default directory.

DECnet-DOS also provides the Transparent Network Task (TNT) utility, which you can use to check for TTT errors. You can also use TNT to remove TTT from memory. Refer to Appendix E for instructions on using the TNT utility.

3.4 Creating a Transparent Communication Task

Once TTT is installed, a high level language task can invoke standard I/O function calls. An assembly language task can use the following MS-DOS function requests.

Table 3-1: MS-DOS Function Requests for Transparent Intertask Communication

Function	Network Access
Create/Open	Initiate a logical link request. Accept a logical link request.
Close	Terminate a logical link connection.
Read	Receive data over a logical link.
Write	Send data over a logical link.

Whether you are running a high level or assembly language task, network access requires the use of specially formatted task names. These task names are implemented as network task specification strings. The strings must be specified with all create and open file operation calls.

3.5 Network Task Specifications

The network task specifications consist of a node specification string (with optional access control information) and a target task specification string. Access control information contains arguments that define your access rights at the remote node. The control string contains three fields: user name, password, and account number. Access control verification is performed according to the conventions of the remote node.

The target task can be identified as either a named or numbered object. Named objects are user-written tasks which are referenced by a name during a connect request and an accept request. The object number for such tasks is 0. Numbered objects are tasks which are referenced by a number. The object numbers range from 1 to 255. Numbers 1 to 127 are reserved for Digital-developed tasks. Numbers 128 to 255 are available for user-written tasks.

You can access the target task by its object name or number. The network task specification string uses one of the following formats:

1. To access the target task by object name with access control information:
`\\t\node\user-id\password\account\object-name`
2. To access the target task by object name without access control information:
`\\t\node\object-name`
3. To access the target task by object number with access control information:
`\\t\node\user\password\account\#object-number`
4. To access the target task by object number without access control information:
`\\t\node\#object-number`
5. To establish a server task by object name:
`\\t\SYS$NET\object-name`
6. To establish a server task by object number:
`\\t\SYS$NET\#object-number`

NOTE

You must specify either a lowercase t or an uppercase T as part of the target task specification string.

3.5.1 Node Specifications

A node specification for a client task names the remote node and supplies optional access control data. The node specification string is preceded by two backslashes, the letter t and another backslash. The optional access control string follows the node information. Each element is separated by a backslash.

The node specification string takes the following format:

```
\\t\node\user\password\account\
```

or

```
\\t\node\
```


where

<i>node</i>	specifies either the name or address of the remote node. A node name has a maximum of 6 alphanumeric characters with at least one alphabetic character. A node address is a numeric string including the area number in the range of 1 to 63, and the node number in the range of 1 to 1023.
<i>user-id</i>	identifies a user name or log-in ID on the remote system. The user name and password set the user's privileges for accessing the remote task. A user name has a maximum of 39 alphabetic characters.
<i>password</i>	defines a user's password that is associated with <i>user</i> . A user's password has a maximum of 39 alphabetic characters.
<i>account</i>	identifies a billing account number which is used with the user name and password information on some systems. An account number has a maximum of 39 characters. If the account information is not required, you can omit it from the string.

A node specification string for a server task is always `\\t\\SYS$NET\\`.

3.5.2 Task Specifications

For a client task, the task specification identifies the cooperating task on the remote system. The server task specification identifies the server task. The task can be specified as a named or a numbered object.

The task specification string is expressed in one of the following formats:

object-name or *#object-number*

where

object-name specifies the task as a named object. The object name has a maximum of 16 characters.

#object-number specifies the task as a numbered object. The valid range is 1 to 255.

3.6 MS-DOS Intercept Routine

Whenever an I/O file operation call is invoked, system control is transferred to the MS-DOS task-to-task intercept routine. Network access is signaled by the string `\\t\` which begins the network task specification string. (See Section 3.5 for formats.) The MS-DOS intercept routine checks to see if the proposed I/O operation is a network supported call.

The intercept routine parses the network task specification string. It stores away the socket numbers for the handles used with the open and create I/O calls. These values must be specified with subsequent read, write and close operations. Before one of these calls can complete, the intercept routine must verify the current status of the network handles.

3.7 Using the Transparent Network Task Control Utility

The Transparent Network Task (TNT) Control utility, Version 2.0, reports the status of the Transparent Task-to-Task (TTT) utility as well as the Transparent File Access (TFA) utility. It features an on-line help routine which lists supported TNT commands. TNT returns extended error information for assisting in fault isolation. Using TNT, you can deinstall TTT (and/or TFA) from memory.

See Appendix E for instructions on using the TNT utility.

3.8 TTT Programming Considerations

There are specific MS-DOS function requests that support DECnet-DOS transparent task-to-task communication. Table 3-1 provides you with a summary of these calls. When creating TTT applications, you should note the following:

- Some user programs may not accept the TTT network specification string.
- You should not use unsupported MS-DOS function calls to perform transparent task-to-task communication.
- If you press `CTRL/C` while TTT is active, network operation may be blocked. To clear this condition, run the TNT utility.

3.9 MS-DOS Function Requests for Transparent Task-to-Task Communication

The following sections describe the MS-DOS function requests and provide specific guidelines.

The function requests are discussed in alphabetical order.

3EH: Close File Handle

Close – terminate a logical link connection and deactivate the original handle.

On Entry:

AH = 3EH

BX = handle for logical link

On Return:

If function succeeded:

Carry bit = clear

If function failed:

Carry bit = set

AX = error code

6 – Invalid handle value detected

DESCRIPTION

The *Close* function request terminates the logical link connection and deactivates the handle used for data exchange. Either task can issue the *Close* call.

On entry, the BX register contains the 16-bit handle value returned by the open or create I/O operation. If the close operation completes successfully and the carry bit is clear, no error is returned in the AX register. If an error condition occurs and the carry bit is set, the appropriate error code is returned in the AX register.

3CH: Create/Open a File

3CH: Create/Open a File

Create/Open – initiate or accept a logical link connection request.

On Entry:

AH = 3CH

DS:DX = address of remote file specification string

On Return:

If function succeeded:

Carry bit = clear

AX = handle for logical link

If function failed:

Carry bit = set

AX = error code

2 – Network process may not be loaded. Node name to node address mapping not found in database file. Target task on the outgoing connection not available. Network is unreachable.

3 – Target task not found.

4 – Too many active logical link connections.

5 – Remote object rejected the request.

DESCRIPTION

In the context of DECnet-DOS, the *Create* and *Open* calls perform the same functions. Either call can initiate and/or accept a logical link connection. However, if *SYS\$NET* is specified as the node name in the network task specification, and supplied with either call, the function is interpreted **only** as the task accepting a logical link connection.

- **To initiate a logical link connection.** The *Create* or *Open* call enables a source task to initiate a logical link connection. On entry, DS:DX contains the address of the network task specification string. Any optional access control information is passed as part of the string to the target task.

On return, the AX register contains an error code or a 16-bit handle associated with the task. The returned handle value must be used for subsequent read and write I/O operations.

- **To accept a logical link connection.** The *Create* or *Open* call accepts a logical link request from another network task. The 16-bit handle value is returned in the AX register. This handle must be used for subsequent read and write operations. On entry, DS:DX contains the address of the network task specification string.

If you are unable to initiate or accept a logical link connection, an error code is returned in the AX register.

To obtain extended error information, run the TNT utility.

3FH: Read from a File

3FH: Read from a File

Read – receive data over a logical link connection.

On Entry:

AH = 3FH
BX = handle for logical link access
DS:DX = address of file data buffer
CX = size of file data buffer

On Return:

If function succeeded:

Carry bit = clear
AX = number of bytes received from file

If function failed:

Carry bit = set
AX = error code

5 – Remote object rejected the request.

6 – Invalid handle detected.

3FH: Read From a File

DESCRIPTION

The *Read* function request allows the target task to receive data sent over the logical link.

On entry, the BX register contains the 16-bit handle value. The CX register contains the number of bytes to be received. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully received by the target task. If an error condition occurs, zero bytes are returned. To obtain extended error information, run the TNT utility. See Appendix D for a list of extended error messages.

40H: Write to a File

Function 40H: Write to a File

Write – send data over a logical link connection.

On Entry:

AH = 40H
BX = handle for logical link access
CX = size of file data buffer
DS:DX = address of file data buffer

On Return:

If function succeeded:

Carry bit = clear
AX = number of bytes written to file

If function failed:

Carry bit = set
AX = error code
5 – Remote object rejected the request.
6 – Invalid handle detected.

40H: Write to a File

DESCRIPTION

The *Write* function request allows the source task to send data over the logical link.

On entry, the BX register contains the 16-bit handle value. The CX register contains the number of bytes to be sent. DS:DX contains the address of the network message buffer.

On return, the AX register contains the number of bytes successfully sent by the source task. If an error condition occurs, the error code is returned in the AX register. To obtain extended error information, run the TNT utility.

Part II



11 1923



C Language Task-to-Task Interface

DECnet-DOSTM includes C language source files which are used to create a linkable library for DECnet-DOS applications. This library provides compatibility with the network socket interface supported by DECnet-ULTRIXTM.

4.1 Creating the DECnet-DOS Programming Interface Library

The file DNETLIB.SRC contains three types of files: the .C files (C language sources), the .H files (header files that contain definitions for the network interface) and the .ASM files (assembly language sources). You should refer to the appropriate installation guide for a complete list of these files.

In order to interface with the DECnet-DOS network process, you need a library against which to link your DECnet-DOS program(s). If you installed DECnet-DOS from floppy diskettes, this library has already been created for you.

If you installed DECnet-DOS from tape (TK50), you must use the following procedure to create a DECnet-DOS programming interface library:

1. The Break Source utility, BREAKSRC, allows you to break the source file, DNETLIB.SRC, into separate source files for compilation and assemblies. BREAKSRC is supplied with the DECnet-DOS distribution kit. If you run the DECnet-DOS Installation Procedure (DIP), you can select to have the DNETLIB.SRC file split into separate files. The BREAKSRC utility will then be run automatically for you. For instructions on how to run DIP, refer to the appropriate installation guide.

To use BREAKSRC manually, follow this format:

```
BREAKSRC <input-file-spec> <output-device: \path>
```

For example:

```
BREAKSRC A:DNETLIB.SRC C:\DECNET\SRC\
```

2. Use your C language compiler to compile each C language source module.
Use your assembler to assemble each assembly source module.

NOTE

If you are using a Microsoft[®] C compiler, you must use the /zp switch when compiling your program.

3. After you produce an object module for each source module, build a library against which to link your DECnet-DOS applications programs.

4.1.1 DECnet-DOS Programming Considerations

The following programming considerations should be noted when writing and developing your DECnet-DOS applications:

1. Using the external variable *errno* — Most DECnet-DOS programming interface functions use the external variable *errno* as a place to return error detail. It is assumed that *errno* has been defined externally to the programming interface as an *int*. It may already be defined in your C language run-time library; if not, your applications program should define it.

Appendix C lists the error codes returned by DECnet-DOS in *errno*.

2. Checking software compatibility — When creating DECnet-DOS applications, make sure that you resolve any C language compiler incompatibilities (such as long variable names or certain type definitions) before compiling the C language source modules.
3. If your C compiler does not do so by default, you should compile the sources so that all data structures are stored without extra space (packed) for alignment of members on “int” boundaries.

4. Using assembly source modules — There are assembly source modules included in DNETLIB.SRC. Before you can successfully call these functions from sources compiled by your C language compiler, you should fulfill any assembly-format requirements (such as segment names). (Refer to the header file, *begin.h*, on the distribution kit as an example of specific C language compiler segment naming requirements.)
5. Using specific macros — There are references to the macros/functions such as *toupper* and *islower* in some of the C language source modules. It is assumed that your C language compiler has provided a standard macro/function for them. If not, you can simply provide your own macros/functions.
6. If you are not using a C compiler that supports the *signal()* handling function (for example, CTRL/C trapping), then you will need either to provide your own *signal()* function or to comment out from the code in *dnet_conn* any references to the *signal()* function and the reference to the include file, *<signal.h>*.

The CTRL/C, provided in *dnet_conn*, allows you to abort a utility which appears to be waiting indefinitely for connections to complete. (This applies only to utilities that use *dnet_conn* for making connections.) The *connect()* function in *dnet_conn* is invoked in a nonblocking socket mode.

If CTRL/C trapping code is commented out from *dnet_conn*, there is the potential problem of leaving hung sockets and running out of system resources, should users decide to enter CTRL/C in the middle of nonblocking connection request attempts.

7. If you are using a compiler that does not define the following variables: *int daylight*, *long timezone* and *char *tzname[2]*, either comment these external declarations out of the header file *<time.h>* or define them in your C program code.
8. The DECnet-DOS programming interface library was based on a 2-segment model: one code segment and one data segment. Three sizes of models are supported: small model, medium model, and large model.

CAUTION

If a program terminates with any active logical links (sockets), the links remain active. In this way, another program can start and use the same links.

If the logical links are not needed, you must issue an *sclose* function call before a program is terminated with an EXIT, `CTRL/Z` or `CTRL/C`. If too many links are left active, an error message, indicating that no more buffers are available or that there are insufficient network resources, is reported. This causes the network to become unusable. To completely deactivate any logical links, run the Network Control Program (NCP) and issue the SET KNOWN LINKS STATE OFF command. (See the *DECnet-DOS Network Management Guide* for more information on this command.)

4.2 How to Read the Socket Interface Call Descriptions

The socket interface calls are presented as separate entries in this manual. They are documented in a consistent manner. Each call is described under the following headings:

NAME	gives the exact name and a brief description of its function.
SYNTAX	shows the complete syntax. Possible call options and the type of expected argument(s) are indicated.
DESCRIPTION	provides more detail on what the call does and how its action is modified by the options.
DIAGNOSTICS	gives explanations of error messages that may be produced.

4.3 Understanding a SYNTAX Section

Each socket interface call documented in this chapter has a SYNTAX entry. It shows how a call is defined. The SYNTAX entry consists of several components.

The SYNTAX section for the *bind* call illustrates these components:

```
int      bind(s, name, namelen)
int      s, namelen;
struct   sockaddr_dn *name;
```

The first line represents the function call and a list of input arguments. Each function call should do one specific task. Data may be passed to the called function by way of arguments. The input arguments follow the function name. They are separated by commas and surrounded by parentheses.

The next set of lines lists the formal arguments and their respective data types (char, int, or structure).

Using the *bind* call example, **name* declares *name* to be a pointer to the structure type *sockaddr_dn*. This structure contains several modifiable fields.

4.4 Socket Function Calls

The following sections describe the socket interface function calls for C programs. They also provide you with specific guidelines. Some of the socket interface calls use specific data structures. Appendix B details how each data structure is formatted.

The socket interface calls are summarized in the following table:

Table 4-1: Socket Interface Calls

Socket Call	Description
accept	Accept an incoming connection request on a socket, and return a socket number.
bind	Assign an object name or number to a socket.
connect	Initiate a connection request on a socket.
getpeername	Get the name of a connected peer on a socket.
getsockname	Get the current name for the specified socket.
getsockopt	Get options associated with sockets.
listen	Listen for pending connections on a socket.
recv	Receive data and out-of-band messages on a socket.
sclose	Terminate a logical link connection and deactivate a socket.
select	Check the I/O status of the network sockets.
send	Send data and out-of-band messages on a socket.
setsockopt	Set options associated with sockets.
shutdown	Shut down part or all of a full duplex logical link connection.
ioctl	Control the operations of sockets.
socket	Create an endpoint for communication and return a socket number.
sread	Read data on a socket.
swrite	Write data to a socket.

4.4.1 Example Socket Interface Calling Sequence

The DECnet-DOS programming interface library was based on a 2-segment model: one code segment and one data segment.

The following program segments illustrate the socket interface calls used by DECnet-DOS client and server tasks.

Socket calls issued by a client task:

```
s = socket(...)          /* get a DECnet socket          */
setsockopt(s,,DSO_CONDATA,,) /* set up optional data      */
setsockopt(s,,DSO_CONACCESS,,) /* set up access            */
                           /* control information        */
connect(s,...)           /* initiate connection to    */
                           /* the server task            */
getsockopt(s,,DSO_CONDATA,,) /* get returned status and   */
                           /* optional data              */
send(s,...)              /* send data (or write)      */
recv(s,...)              /* receive data (or read)    */
setsockopt(s,,DSO_DISDATA) /* set up optional          */
                           /* data for disconnect        */
sclose(s)                /* terminate connection      */
```

Socket calls issued by a server task:

```
s = socket(...)          /* get a DECnet socket          */
bind(s,...)              /* bind a name to the socket   */
listen(s,...)            /* make the socket available   */
                           /* for client connections      */
setsockopt(s,,DSO_ACCMODE, /* accept mode,               */
           ACC_DEFER,)    /* deferred/immediate          */
ns = accept(s,...)        /* await connect(s) on the    */
                           /* new socket, ns              */
getsockopt(ns,,DSO_CONACCESS,,) /* get access                  */
                           /* control information        */
getsockopt(ns,,DSO_CONDATA,,) /* get optional connect data   */
setsockopt(ns,,DSO_CONDATA,,) /* set up optional data        */
setsockopt(ns,,DSO_CONACCEPT,,) /* finally accept the         */
                           /* connection request          */
recv(ns,...)             /* receive data (or read)     */
send(ns,...)             /* send data (or write)       */
setsockopt(ns,,DSO_DISDATA,,) /* set up optional            */
                           /* disconnect data             */
sclose(ns)               /* terminate connection       */
sclose(s)                /* terminate DECnet path      */
```

accept

accept

NAME

`accept` — accept an incoming connection request on a socket and return a socket number.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>

Int      accept(s, sorcbk, sorclen)
Int      s, *sorclen;
struct   sockaddr_dn *sorcbk;
```

DESCRIPTION

The *accept* call extracts the first connection request on the queue of pending connections, creates a new socket with a new number having the same properties as the original listening socket. The original socket remains opened.

If the socket is set to nonblocking I/O, and there are no queued connection requests, *io_status* will return a -1 and *errno* will contain EWOULDBLOCK.

There are two modes of accepting an incoming connection. They are immediate and deferred modes. These modes of acceptance are set by using the *setsockopt* call. When immediate mode is in effect, the connection is established immediately. The deferred mode indicates that the server task completes the *accept* call without fully completing the connection to the client task. In this case, the server task can examine the access control or optional data before it decides to accept or reject the connection request. The server task can then issue the *setsockopt* call with the appropriate reject or accept option.

Input Arguments

- s* specifies the number for a socket that was created with the *socket* call, bound to a name or number by the *bind* call, and was set to listen for connects by the *listen* call.
- sorcbk* is a value result argument. It specifies an address of a structure *sorcbk* of the data type *sockaddr_dn*. This argument will be filled in with the information of the entity requesting the connection.
- sorclen* is a value result argument. It specifies the address of an int. The value of *sorclen* should initially contain the size of the *sorcbk*.

Return Arguments

- sorclen* specifies the actual length of the returned data in bytes.
- sorcbk* specifies the socket address data structure, *sockaddr_dn*. A user retrieves data from the fields filled in by this function call. (See Appendix B on how *sockaddr_dn* is formatted.)

The following data fields are filled in by this function call:

- sdn_family* is the address family AF_DECnet.
- sdn_objnum* is the object number for the client task. It can be a number 0 to 255. It is set to 0 only when the object name is used.
- sdn_objnamel* is the size of the object name.
- sdn_objname* is the object name of the client task. It can be up to a 16-element array of char. It is used only when *sdn_objnum* equals 0.
- sdn_add* is the node address structure for the client task. (See Appendix B on how *dn_naddr* is formatted.)

accept

Return Value

If the call succeeds, it returns a nonnegative integer called a socket number. This number will be used for communications over a logical link connection. If an error occurs, the call returns a -1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[ECONNABORTED]	The client task disconnected before the <i>accept</i> call completed.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENFILE]	There are no more available sockets.
[EWOULDBLOCK]	The socket is marked for nonblocking and no connections are waiting to be accepted.

bind

NAME

bind – assign an object name or number to a socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
int      bind(s, name, namelen)
int      s, namelen;
struct   sockaddr_dn *name;
```

DESCRIPTION

The *bind* call assigns an object name or number to a socket. When a socket is first created with the *socket* call, it exists in a name space but has no assigned name or number. The *bind* call is used primarily by server tasks. The object name is required before a server task can listen for incoming connection requests using the *listen* call. It can also be used by client tasks to identify themselves to server tasks. See also the *accept*, *connect*, *getpeername*, and *getsockname* calls.

NOTE

VMS proxy access by user name is made possible if the client task uses the *bind* call specifying a user name as the object name. Refer to the `SO_REUSEADDR` option for the *setsockopt* call if you want to make more than one proxy connection with the same name.

bind

Input Arguments

<i>s</i>	specifies the number for a socket that has been created with the <i>socket</i> call.
<i>name</i>	specifies the address of the structure <i>name</i> of data type <i>sockaddr_dn</i> . A user fills in the data for each field. (See Appendix B on how <i>sockaddr_dn</i> is formatted.)

The following data fields can be modified:

<i>sdn_family</i>	specifies the address family as AF_DECnet.
<i>sdn_flags</i>	specifies the object flag option. It must be set to 0.
<i>sdn_objnum</i>	defines the object number for the server task. It can be a number 0 to 255. It is set to 0 only when the object name is used.
<i>sdn_objnamel</i>	is the size of the object name.
<i>sdn_objname</i>	defines the object name of the server or client task. It can be up to a 16-element array of char. It is used only when <i>sdn_objnum</i> equals 0.
<i>sdn_add</i>	specifies the node address structure for the server task. This data member is ignored.
<i>namelen</i>	specifies the size of the name structure.

Return Value

If the bind is successful, a 0 value is returned. An unsuccessful bind returns a value of -1. When an error condition exists, the external variable *errno* will contain error detail.

DIAGNOSTICS

[EADDRINUSE]	The specified name or number is already used by another socket.
[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EINVAL]	The socket <i>s</i> is already bound to a name or number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

connect

connect

NAME

connect – initiate a connection request on a socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
int      connect(s, destblk, destlen)
int      s, destlen;
struct   sockaddr_dn, *destblk;
```

DESCRIPTION

The *connect* call issues a connection request to another socket. The other socket is specified by *destblk* which is a pointer to the *destblk* data structure.

Optional data as well as access control information may be passed with this function call. This data must be previously set by the *setsockopt* call. If subsequent *connect* calls are issued on the same socket, a task must reissue the *setsockopt* call to set up new optional user data and/or access control information.

Input Arguments

<i>s</i>	specifies the number for the socket which has been created with the <i>socket</i> call. This socket number is used for establishing a connection between the user tasks. It is also used with subsequent send and receive function calls.
<i>destblk</i>	specifies the address of the structure <i>destblk</i> of the data type <i>sockaddr_dn</i> . A user fills in the data for each field. (See Appendix B on how <i>sockaddr_dn</i> is formatted.)

The following data fields can be modified:

<i>sdn_family</i>	specifies the address family as AF_DECnet.
-------------------	--

<i>sdn_flags</i>	specifies the object flag option. It must be set to 0.
<i>sdn_objnum</i>	defines the object number for the server task. It can be a number 0 to 255.
<i>sdn_objname</i>	is the size of the object name.
<i>sdn_objname</i>	defines the object name of the server task. It can be up to a 16-element array of char. It is used only when <i>sdn_objnum</i> equals 0.
<i>sdn_add</i>	specifies the node address structure for the server task. (See Appendix B on how <i>dn_naddr</i> is formatted.)
<i>destlen</i>	specifies the size of the destination block structure.

Return Value

If the call succeeds, it returns a value of 0. Otherwise, the call returns a value of -1. When an error condition exists, the external variable *errno* will contain error detail. If the socket is set to nonblocking I/O (see also *siocfl*), and you issue a *connect*, the function returns a -1, and the error message EINPROGRESS.

DIAGNOSTICS

[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this particular socket.
[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EBUSY]	The socket is not in idle state. The socket is in the process of being connected or disconnected; it is currently a connected or listening socket.
[ECONNABORTED]	The peer task has disconnected and the connection was aborted.
[ECONNREFUSED]	The attempt to connect was forcefully rejected.
[ECONNRESET]	The remote task has disconnected the link.

connect

[EHOSTUNREACH]	The remote node is unreachable.
[EINPROGRESS]	The connection request is now in progress.
[ENETDOWN]	The network is down. The Executor name and address may not have been set and/or the Executor state may not have been set ON.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ERANGE]	The object number of the server task is invalid. The valid range is 0 to 255.
[ESRCH]	The server object does not exist on the remote node.
[ETIMEDOUT]	Connection establishment was timed out before a connection was established. The remote node may not be on the network.
[ETOOMANYREFS]	The remote node has accepted the maximum number of connection requests.
[ENETRESET]	The remote node aborted the link due to a link mismatch, a system reboot, or most probably, a configuration error.

getpeername

NAME

getpeername – get the name of a connected peer on a socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
Int      getpeername(s, destblk, destlen)
Int      s, *destlen;
struct   sockaddr_dn *destblk;
```

DESCRIPTION

The *getpeername* call returns information about the peer socket connected to the specified socket. This information is the same information returned by the *accept* call. It may be used by a client or server task anytime after a connection has been established between two tasks or peers.

Input Arguments

<i>s</i>	specifies the number for a socket which has been created by the <i>socket</i> or the <i>accept</i> call.
<i>destblk</i>	specifies the address of the <i>destblk</i> structure of the data type <i>sockaddr_dn</i> . This argument will be filled in with peer information returned by <i>getpeername</i> .
<i>destlen</i>	is a value result argument. It specifies the address of an int. The value of <i>destlen</i> should be initialized to the size of the <i>destblk</i> .

getpeername

Return Arguments

<i>destlen</i>	specifies the actual size of the destination block (in bytes).
<i>destblk</i>	specifies the socket address data structure, <i>sockaddr_dn</i> . A user retrieves data from the fields filled in by this function call. (See Appendix B on how <i>sockaddr_dn</i> is formatted.)

The following data fields can be filled in by this function call:

<i>sdn_family</i>	is the address family AF_DECnet.
<i>sdn_objnum</i>	is the object number for the peer task. It can be a number 0 to 255.
<i>sdn_objname1</i>	is the size of the object name.
<i>sdn_objname</i>	is the object name of the peer task. It can be up to a 16-element array of char. It is only used when <i>sdn_objnum</i> equals 0.
<i>sdn_add</i>	is the address structure for the peer node. (See Appendix B on how <i>dn_naddr</i> is formatted.)

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, the call returns a -1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The socket <i>s</i> is not connected, it has no peer.

getsockname

NAME

getsockname – get the current object name or number for the specified socket.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
Int      getsockname(s, destblk, destlen)
Int      s, *destlen;
struct   sockaddr_dn *destblk;
```

DESCRIPTION

The *getsockname* call returns the bound object name or number of the specified socket.

Input Arguments

<i>s</i>	specifies the number for a socket which has been created by the <i>socket</i> or the <i>accept</i> call.
<i>destblk</i>	specifies the address of a structure of the data type <i>sockaddr_dn</i> . This argument will be filled in with local task information returned by <i>getsockname</i> .
<i>destlen</i>	is a value result argument. It specifies the address of an int. The value of <i>destlen</i> should be initialized to the size of the <i>destblk</i> .

Return Arguments

<i>destlen</i>	specifies the actual size of the destination block (in bytes).
<i>destblk</i>	specifies the socket address data structure, <i>sockaddr_dn</i> . A user retrieves data from the fields filled in by this function call. (See Appendix B on how <i>sockaddr_dn</i> is formatted.)

getsockname

The following data fields can be filled in by this function call:

- sdn_family* is the address family AF_DECnet.
- sdn_objnum* is the object number for the local task. It can be a number 0 to 255.
- sdn_objnamel* is the size of the object name.
- sdn_objname* is the object name of the local task. It can be up to a 16-element array of char. It is only used when *sdn_objnum* equals 0.
- sdn_addr* is the address structure for the local node. (See Appendix B on how *dn_naddr* is formatted.)

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, the call returns a -1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

- [EBADF] The argument *s* does not contain a valid socket number.
- [ENETUNREACH] The network is unreachable. The network process is not installed.

getsockopt

NAME

getsockopt — get the options associated with sockets.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
Int      getsockopt(s, level, optname, optval, optlen)
Int      s, level optname, *optlen;
char     *optval;
```

DESCRIPTION

The *getsockopt* call manipulates various options associated with a socket. Options exist at multiple levels and you must specify the level number for the desired operation.

At the socket level (SOL_SOCKET), the options include:

- **SO_KEEPALIVE.** If this option is set on a socket, any links and sockets associated with this socket will remain active, despite any attempts to disconnect them.

NOTE

Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *setsockopt* call with SO_KEEPALIVE turned off. To turn off SO_KEEPALIVE, you must precede SO_KEEPALIVE with a tilde (~), as in, ~SO_KEEPALIVE. (~SO_KEEPALIVE is the default condition.)

getsockopt

You then issue the *sclose* call. The logical links (if any) are disconnected, and the socket and associated sockets (if any) are deallocated. However, if you issue *sclose* without turning off `SO_KEEPAIVE`, the sockets remain allocated, and the links (if any) stay active.

- **SO_LINGER** controls the actions taken when unsent messages are queued on a socket and the *sclose* call is issued. If `SO_LINGER` is set, the connection is maintained until the outstanding messages have been sent. This is the default condition.
- **SO_DONTLINGER** also controls the actions of unsent messages. If `SO_DONTLINGER` is set, and the *sclose* call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.
- **SO_REUSEADDR** allows the reuse of a name already bound to a socket. For most situations, a name is bound to a socket only once. However, this option enables you to reuse the same name. This particular option **must** be used only for outgoing connection requests. It cannot be used for incoming connections.

VMS proxy access by user name is made possible if the client task uses the `BIND` call specifying the user name as the object name. If you wish to make more than one proxy connection with the same user name, you must use the `SO_REUSEADDR` option.

- **SO_RCVUSRBUF** tells DNP that the user is using `MSG_USRBUF` receives and that select functions (with `readfds` selected) should complete upon receipt of the first segment of a message. Otherwise, *select()* does not indicate that the message is read-ready until the end of the message is received.

At the DECnet level (DNPROTO_NSP), socket options may specify the way in which a connection request is accepted or rejected, may be used to set up optional user data and/or access control information, or may be used to obtain current link state information. The following socket options can be specified:

- **DSO_ACCEPTMODE**. The accept option mode is used at the DECnet level for processing *accept* calls. A socket must be bound (see *bind*) before specifying this option. There are three values which can be supplied for this option. They are immediate mode, **ACC_IMMED**, deferred mode, **ACC_DEFER**, and reuse mode, **ACC_REUSE**.
 - **ACC_IMMED** mode is the default condition for this option. When immediate mode is in effect, control is immediately returned to the server task following an *accept* call with the connection request accepted. The access control information and/or optional data is ignored by the server task.
 - **ACC_DEFER** mode enables the server task to complete the *accept* call without fully completing the connection to the client task. In this case, the server task can examine the access control or optional data before it decides to accept or reject the connection request. The server task can then issue the *setsockopt* call with the appropriate reject or accept option.
 - **ACC_REUSE** enables your application to reuse the listening socket to receive incoming connections. To receive the connect, your application must issue an **ACCEPT** call. Your application must supply the socket number (*iocb.io_socket*) either as the same value as the listening socket or as a zero that will return the correct socket.

This mode can be used along with **ACC_IMMED** or **ACC_DEFER**. The argument to **DSO_ACCEPTMODE** should be the OR of the values (such as **ACC_DEFER | ACC_REUSE**). If the connection is rejected or disconnected, the socket must be explicitly reenabled for listening by issuing a new **LISTEN** call.

getsockopt

NOTE

This call is intended primarily for use by single-threaded servers. From the time a single connect is received on the socket until another listen is posted, further connects are rejected with the message "Object unknown." Another listen must be posted quickly, but that does not close the timing window. The client must be aware of the possible rejection and retry.

- **DSO_CONDATA** allows up to 16 bytes of optional user data to be set by the *setsockopt* call. It can be sent as a result of the *connect* or the *accept* (with the deferred option) calls. The optional data is passed in a structure of type *optdata_dn*. (See Appendix B on how *optdata_dn* is formatted.) The data is read by the task issuing the *getsockopt* call with this option.
- **DSO_DISDATA** allows up to 16 bytes of optional data to be set by the *setsockopt* call. It can be sent as a result of the *sclose* call. The optional data is passed in a structure of type *optdata_dn*. (See Appendix B on how *optdata_dn* is formatted.) The data is read by the task issuing the *getsockopt* call with this option.
- **DSO_CONACCESS** allows access control information to be passed by the user task. This information is set with the *setsockopt* call. The access data is sent to the server task. It is passed with the *connect* call in a structure of type *accessdata_dn*. (See Appendix B on how *accessdata_dn* is formatted.) The access data is read by the task issuing the *getsockopt* call with this option.
- **DSO_LINKINFO** determines the state of the logical link connection. When the *getsockopt* call is issued with this option, the state of the logical link is returned in a logical link information data structure, *linkinfo_dn*. (See Appendix B on how *linkinfo_dn* is formatted.)

Input Arguments

<i>s</i>	specifies the number for a socket returned by the <i>socket</i> or the <i>accept</i> call.
<i>level</i>	specifies the level at which options are manipulated. The level is either SOL_SOCKET or DPROTO_NSP. (See Appendix A for details.)
<i>optname</i>	specifies options to be interpreted at the level specified. For example, SO_LINGER at the SOL_SOCKET level.
<i>optval</i> , <i>optlen</i>	specify access option values. The interpretation of each argument is function dependent as shown here:
<i>optval</i>	specifies the address of a buffer which will contain the returned value for the requested option(s).
<i>optlen</i>	is a value result parameter. It specifies the address of an int. The value of <i>optlen</i> should initially contain the size of the buffer pointed to by <i>optval</i> . On return, it will contain the actual size of the returned value. Note that getsockopt passes the address of the length of this value.

Output Arguments

<i>optval</i>	specifies the buffer which contains the returned value for the requested socket option(s).
<i>optlen</i>	specifies the actual size of the returned value.

Return Values

If the call completes successfully, a value of 0 is returned. An unsuccessful call returns a value of -1. When an error condition exists, the external variable *errno* will contain error details. See the DIAGNOSTICS section for a full description of the error messages.

getsockopt

DIAGNOSTICS

[EACCES]	Unable to disconnect the socket.
[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[ECONNABORTED]	The accept connect did not complete. The peer task disconnected and the connection was aborted.
[EDOM]	The acceptance mode is not valid.
[EFAULT]	The network received an invalid buffer.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOBUFS]	There are no available buffers for optional access control and/or user data.
[ENOPROTOOPT]	No access control information was supplied with the connection request.
[EOPNOTSUPP]	The option is unknown.

listen

NAME

listen – listen for pending connections on a socket.

SYNTAX

```
int      listen(s, backlog)
int      s;
int      backlog;
```

DESCRIPTION

The *listen* call declares your socket as a server which is available for client connections. The server uses the bound name or number in order to listen for incoming client connections. This call must be issued before an incoming connection can be accepted or rejected. See also the *accept*(), the *bind*(), and the *select*() calls.

If you detach a listening socket while the socket is receiving client connections, then all links associated with the listening socket immediately abort and all outstanding data is lost.

Input Arguments

<i>s</i>	specifies the number for a socket which has been created with the <i>socket</i> call and bound to a name or number by the <i>bind</i> call.
<i>backlog</i>	defines the maximum number of unaccepted incoming connects which are allowed on this particular socket. If a connection request arrives when the queue is full, the client task will receive an error with an indication of ETOOMANYREFS.

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, the call returns a -1. When an error condition exists, the external variable *errno* will contain error detail. See the DIAGNOSTICS section for a full description of the error messages.

listen

DIAGNOSTICS

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[ETOOMANYREFS]	The connection request was rejected, queue is full.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EOPNOTSUPP]	The socket type does not support the listen operation.

NOTE

You may issue a *listen(s, 0)* call while already processing data over a previously accepted connected socket. If you do this, subsequent incoming connection requests will be rejected by the network process. When communications are completed over the currently connected socket, you should reissue the *listen(s, backlog)* call to allow for subsequent acceptance of incoming connection requests.

recv

NAME

recv – receive data or out-of-band messages on a socket.

SYNTAX

#Include <socket.h>

```
int      recv(s, buffer, buflen, flags)
int      s, buflen, flags;
char     *buffer;
```

DESCRIPTION

The *recv* call is used to receive data from your peer. See also the *sread* call.

If no messages are available at the socket, the *recv* call waits for a message to arrive unless the socket is nonblocking. (See *sioctl*.) In this case, a status of *-1* is returned with the external variable *errno* set to *EWOULDBLOCK*.

If the link is disconnected, queued data can still be received on the socket. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *recv* call returns zero bytes.

The *select* call may be used to determine when more data has arrived. (See *select*.)

Out-of-band messages are delivered to a receiving task ahead of normal data messages. These messages can be received by specifying *MSG_OOB* as the flag argument.

recv

Input Arguments

<i>s</i>	specifies the number for a socket returned by the <i>socket</i> or the <i>accept</i> call.
<i>buffer</i>	specifies the address of a buffer which will contain the received message.
<i>buflen</i>	specifies the size of the message buffer.
<i>flags</i>	set to 0 indicates that the task will receive normal messages. If set to <i>MSG_OOB</i> , the task will receive out-of-band messages. Only one out-of-band message can be outstanding at any time. You can also set the <i>flags</i> argument to <i>MSG_PEEK</i> to read the next pending message without removing it from the receive queue.

Output Argument

<i>buffer</i>	specifies the buffer which contains the received message.
---------------	---

Return Value

If the call succeeds, the number of received characters is returned.

If the call returns a zero, you have either received a zero length message or the logical link has been disconnected. To determine the state of the logical link, use the *getsockopt* function call with the *DSO_LINKINFO* option (see *getsockopt*), or the DECnet utility function, *dnet_eof* (see Chapter 5). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

When receiving **normal data**, the following set of error messages can occur:

Blocking I/O Message

Description

[EBADF]

The argument *s* does not contain a valid socket number.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

Nonblocking I/O Message

Description

[EBADF]

The argument *s* does not contain a valid socket number.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

[EWOULDBLOCK]

The receive operation would block because there is currently no data to receive.

When receiving **out-of-band data**, the following set of error messages can occur:

Blocking I/O Message

Description

[EBADF]

The argument *s* does not contain a valid socket number.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

[EWOULDBLOCK]

The receive operation would block because there is currently no data to receive.

recv

Nonblocking I/O Message

Description

[EBADF]

The argument *s* does not contain a valid socket number.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

[EWOULDBLOCK]

The receive operation would block because there is currently no data to receive.

sclose

NAME

sclose – terminate a logical link connection and deactivate a socket.

SYNTAX

```
int sclose(s)
int s;
```

DESCRIPTION

The *sclose* call terminates an outstanding connection over the socket referenced by *s*. It also deactivates the socket.

NOTE

Before you can terminate a connection over a socket with the option `SO_KEEPALIVE` set, you must first issue a *setsockopt* call with `SO_KEEPALIVE` turned off. To turn off `SO_KEEPALIVE`, you must precede `SO_KEEPALIVE` with a tilde (~), as in, `~SO_KEEPALIVE`. (`~SO_KEEPALIVE` is the default condition.)

You then issue the *sclose* call. The logical links (if any) are disconnected, and the socket and associated sockets (if any) are deallocated. However, if you issue *sclose* without turning off `SO_KEEPALIVE`, the sockets remain allocated, and the links (if any) stay active.

The effect of *sclose* on unsent data queued for a remote task depends on the `linger` option set with the *setsockopt* function call. If `SO_LINGER` is set, control is returned to the task, but the link is not disconnected until the unqueued data is sent. If `SO_DONTLINGER` is set, control is returned to the task, and any unqueued data is lost.

sclose

NOTE

The DECnet-DOS function call *sclose* is not compatible with DECnet-ULTRIX systems. See Appendix H for information on how to transport DECnet-DOS programs that use *sclose*.

Input Argument

s specifies the number for a socket which was returned by the *socket* or the *accept* call.

Return Value

If the call succeeds, a value of 0 is returned. If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

- | | |
|---------------|---|
| [EBADF] | The argument <i>s</i> does not contain a valid socket number. |
| [ENETUNREACH] | The network is unreachable. The network process is not installed. |

select

NAME

select – check the I/O status of the network sockets.

SYNTAX

```
#include <time.h>
```

```
int      select(nfds, readfds, writefds, exceptfds, timeout)
int      nfds;
unsigned long *readfds, *writefds, *exceptfds;
struct    timeval *timeout;
```

DESCRIPTION

The *select* call checks the network sockets specified by the bit masks *readfds*, *writefds*, and *exceptfds*, respectively, to see if they are ready for reading, writing, or have any outstanding out-of-band messages. The *select* call does not tell you if the logical link connection has been broken.

You should use the *select* call to help manage your *accept*, *send*, *recv*, *swrite*, and *sread* calls.

The *readfds*, *writefds*, and *exceptfds* I/O descriptors are long words which contain bit masks. Each bit in a mask represents one socket number. For example, socket “3” is the fourth bit or has a hex value of 8.

NOTE

The *select* call can check socket numbers only in the range 0 to 31.

To specify the bit for any socket number, use the value returned by the *socket* or the *accept* call, as “1<<*s*”.

select

Input Arguments

nfds specifies the highest socket number to be checked. The bits from $(1 \ll 0)$ to $(1 \ll (nfds - 1))$ are examined.

readfds specifies the socket numbers to be examined for read ready. For listening sockets, a read ready condition indicates that an incoming connection request can be read and either accepted or rejected. For sequenced sockets, there is a complete message to be read. For stream sockets, there is some data to be read. If a socket disconnects or aborts, a read ready condition will always occur.

NOTE

To prevent a program from hanging on a stream socket, issue the *sioctl* call with the FIONREAD function argument (see *sioctl*), and then read only those numbers of bytes returned by the call. You should also perform socket operations in nonblocking I/O mode.

This descriptor can be given as a null pointer if of no interest.

writefds specifies the socket numbers to be examined for write ready. A write ready condition exists when the logical link is available. This descriptor can be passed as a null pointer if of no interest.

exceptfds specifies the socket numbers to be examined for out-of-band data ready. There is a pending out-of-band message to receive. This descriptor can be given as a null pointer if of no interest.

NOTE

The bit mask *exceptfds* is presently not supported by DECnet-ULTRIX.

select

timeout specifies a pointer to a data structure of type *timeval*. If this pointer is null, then the *select* call will wait until an event occurs. If the pointer is non-null, and the time value is greater than zero, then the *select* call will return either after *n* seconds have expired or when an event occurs, whichever one comes first. If the pointer is non-null and the time value is zero, then the *select* call will return after an immediate poll.

timeval specifies the amount of time to wait. The data members are:

tv sec specifies the time in seconds.

tv usec this data member is ignored.

Output Arguments

read_fds If a socket is read ready, the bit is returned “on”, and *read_fds* returns the socket numbers (as bit masks) to be examined. If the socket is not read ready, the bit is cleared.

write_fds If a socket is write ready, the bit is returned “on”, and *write_fds* returns the socket numbers (as bit masks) to be examined. If the socket is not write ready, the bit is cleared.

except_fds If the socket is out-of-band data ready, the bit is returned “on”, and *except_fds* returns the socket numbers (as bit masks) to be examined. If the socket is not out-of-band data ready, the bit is cleared.

Return Value

The value returned by the *select* call is the number of bits set in all the masks. The bit masks contain the set bits that correspond to the sockets in which events have occurred. If the time period expires, a value of 0 is returned.

If an error occurs, a value of -1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

select

DIAGNOSTICS

[EBADF]

One of the specified bit masks is an invalid descriptor.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

send

NAME

`send` – send data or out-of-band messages on a socket.

SYNTAX

```
#Include <socket.h>
```

```
Int      send(s, buffer, buflen, flags)
Int      s, buflen, flags;
char     *buffer;
```

DESCRIPTION

The *send* call is used to transmit data to your peer. The client task uses the socket number returned by the *socket* call. The server task uses the socket number returned by the *accept* call.

If you cannot get enough buffer space while building the outgoing message on a blocking socket, the message is blocked. You must wait until current transmissions are finished. For a nonblocking socket, the error message, *EWouldBlock*, is returned. If a socket disconnects, any outstanding data to be sent is discarded.

The flag option, *MSG_OOB*, can be set to indicate that out-of-band data will be sent to your peer socket. An out-of-band message is a high priority message that you can send to your peer. This message bypasses any normal messages waiting to be received. An out-of-band message must be received by your peer before another message can be sent.

The *select* call can be used to determine if it is possible to send more data.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

buffer specifies the address of the buffer which contains the outgoing message.

send

- buflen* specifies the size of the outgoing message.
- flags* can be set to 0 to indicate normal messages. It can be set to *MSG_OOB* for out-of-band messages.

Return Value

If the call succeeds, the number of characters sent is returned. If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable *errno*. See the **DIAGNOSTICS** section for a full description of the error messages.

DIAGNOSTICS

When sending **normal data**, the following set of error messages can occur:

Blocking I/O Message

Description

- | | |
|---------------|---|
| [EBADF] | The argument <i>s</i> does not contain a valid socket number. |
| [EMSGSIZE] | The size of the outgoing message is more than 4096 bytes. |
| [ENETUNREACH] | The network is unreachable. The network process is not installed. |
| [ENOTCONN] | The <i>send</i> call did not complete and the link was disconnected. |
| [EPIPE] | The link has been disconnected, aborted, or shut down. No further messages can be sent. |

Nonblocking I/O Message

Description

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 4096 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The <i>send</i> call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.
[EWOULDBLOCK]	The outbound quota was full, and the message could not be sent. Try again later.

When sending **out-of-band data**, the following set of error messages can occur:

Blocking I/O Message

Description

[EALREADY]	The out-of-band message could not be sent. A similar transmission request is still in progress.
[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 16 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The <i>send</i> call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

send

Nonblocking I/O Message

Description

[EALREADY]

The out-of-band message could not be sent. A similar transmission request is still in progress.

[EBADF]

The argument *s* does not contain a valid socket number.

[EMSGSIZE]

The size of the outgoing message is more than 16 bytes.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

[ENOTCONN]

The *send* call did not complete and the link was disconnected.

[EPIPE]

The link has been disconnected, aborted or shut down. No further messages can be sent.

setsockopt

NAME

setsockopt — set the options associated with sockets.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
int      setsockopt(s, level, optname, optval, optlen)
int      s, level, optname, optlen;
char     *optval;
```

DESCRIPTION

The *setsockopt* call manipulates various options associated with a socket. Options exist at multiple levels and you must specify the level number for the desired operation.

At the socket level (SOL_SOCKET), the options include:

- **SO_KEEPALIVE.** If this option is set on a socket, any links and sockets associated with this socket will remain active, despite any attempts to disconnect them.

NOTE

Before you can terminate a connection over a socket with the option SO_KEEPALIVE set, you must first issue a *setsockopt* call with SO_KEEPALIVE turned off. To turn off SO_KEEPALIVE, you must precede SO_KEEPALIVE with a tilde (~), as in, ~SO_KEEPALIVE. (~SO_KEEPALIVE is the default condition.)

setsockopt

You then issue the *sclose* call. The logical links (if any) are disconnected, and the socket and associated sockets (if any) are deallocated. However, if you issue *sclose* without turning off `SO_KEEPAVIVE`, the sockets remain allocated, and the links (if any) stay active.

- **SO_LINGER** controls the actions taken when unsent messages are queued on a socket and the *sclose* call is issued. If `SO_LINGER` is set, the connection is maintained until the outstanding messages have been sent. This is the default condition.
- **SO_DONTLINGER** also controls the actions of unsent messages. If `SO_DONTLINGER` is set, and the *sclose* call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.
- **SO_REUSEADDR** allows the reuse of a name already bound to a socket. For most situations, a name is bound to a socket only once. However, this option enables you to reuse the same name. This particular option **must** be used only for outgoing connection requests. It cannot be used for incoming connections.

VMS proxy access by user name is made possible if the client task uses the `BIND` call specifying the user name as the object name. If you wish to make more than one proxy connection with the same user name, you must use the `SO_REUSEADDR` option.

- **SO_RCVUSRBUF** tells DNP that the user is using `MSG_USRBUF` receives and that select functions (with `readfds` selected) should complete upon receipt of the first segment of a message. Otherwise, select does not indicate that the message is read-ready until the end of the message is received.

At the DECnet level (DNPROTO_NSP), socket options may specify the way in which a connection request is accepted or rejected, may be used to set up optional user data and/or access control information, or may be used to obtain current link state information. The following socket options can be specified:

- **DSO_ACCEPTMODE.** The accept option mode is used at the DECnet level for processing *accept* calls. A socket must be bound (see *bind*) before specifying this option. There are three values which can be supplied for this option. They are immediate mode, **ACC_IMMED**, deferred mode, **ACC_DEFER**, and reuse mode, **ACC_REUSE**.
 - **ACC_IMMED** mode is the default condition for this option. When immediate mode is in effect, control is immediately returned to the server task following an *accept* call with the connection request accepted. The access control information and/or optional data is ignored by the server task.
 - **ACC_DEFER** mode enables the server task to complete the *accept* call without fully completing the connection to the client task. In this case, the server task can examine the access control or optional data before it decides to accept or reject the connection request. The server task can then issue the *setsockopt* call with the appropriate reject or accept option.
 - **ACC_REUSE** enables your application to reuse the listening socket to receive incoming connections. To receive the connect, your application must issue an **ACCEPT** call. Your application must supply the socket number (*iocb.io_socket*) either as the same value as the listening socket or as a zero that will return the correct socket.

This mode can be used along with **ACC_IMMED** or **ACC_DEFER**. The argument to **DSO_ACCEPTMODE** should be the OR of the values (such as **ACC_DEFER | ACC_REUSE**). If the connection is rejected or disconnected, the socket must be explicitly reenabled for listening by issuing a new **LISTEN** call.

setsockopt

NOTE

This call is intended primarily for use by single-threaded servers. From the time a single connect is received on the socket until another listen is posted, further connects are rejected with the message "Object unknown." Another listen must be posted quickly, but that does not close the timing window. The client must be aware of the possible rejection and retry.

- **DSO_CONACCEPT** allows the server task to accept the pending connection on the socket returned by the *accept* call. The original listening socket was set to deferred accept mode. Any optional data previously set by **DSO_CONDATA** will also be sent.
- **DSO_CONREJECT** allows the server task to reject the pending connection on the socket returned by the *accept* call. The original listening socket was set to deferred accept mode. Any optional data previously set by **DSO_DISDATA** will also be sent. The reject reason is the value passed with this option.
- **DSO_CONDATA** allows up to 16 bytes of optional user data to be set by the *setsockopt* call. It can be sent as a result of the *connect* or the *accept* (with the deferred option) calls. The optional data is passed in a structure of type *optdata_dn*. (See Appendix B on how *optdata_dn* is formatted.) The data is read by the task issuing the *getsockopt* call with this option.
- **DSO_DISDATA** allows up to 16 bytes of optional data to be set by the *setsockopt* call. It can be sent as a result of the *sclose* call. The optional data is passed in a structure of type *optdata_dn*. (See Appendix B on how *optdata_dn* is formatted.) The data is read by the task issuing the *getsockopt* call with this option.

- **DSO_CONACCESS** allows access control information to be passed by the user task. This information is set with the *setsockopt* call. The access data is sent to the server task. It is passed with the *connect* call in a structure of type *accessdata_dn*. (See Appendix B on how *accessdata_dn* is formatted.) The access data is read by the task issuing the *getsockopt* call with this option.
- **DSO_FLOWCTRL** allows the DECnet Transport protocol (NSP) to use flow control techniques. Flow control can have a major effect on application performance, and is dependent on how the application uses the network and the basic performance of the systems involved. Two settings are available for **DSO_FLOWCTRL**: NoFlow or Segment Count. The following values apply to each setting:

0 = NoFlow (also known as XON/XOFF or SEND/DONTSEND)
1 = Segment Count (this is the default)

DSO_FLOWCTRL can only be set before a socket has been connected, or, if receiving a connect, before the connect has been accepted. If you set this option on a listening socket, it will be passed on to new connects. An **EBUSY** error is returned if the socket is in the wrong state. There is no *getsockopt()* support for this option.

The **DNP /FC:** switch sets the default flow control for all **NETBIOS** sockets. **NETBIOS** sockets do not have a time period in which options can be adjusted. The **PCSA MS-NET File Services** protocol (**SMB**) is also a request-response protocol, and performs better in most cases with the **/FC:** switch set to zero (**/FC:0**).

Segment Count is the default **DECnet-DOS** technique and works well under adverse conditions. NoFlow works well with request-response protocols or on systems with robust network devices.

setsockopt

Under Segment Count, the receiver sends a count of segments that the transmitter is allowed to send. More credits are sent only when the application receives these segments. Each time credits are sent, the process involves sending another NSP link service message which also must be acknowledged by the other system. Due to restrictions in implementation, DECnet-VAX is incapable of carrying the link service acknowledgement on top of the normal data channel, and generates a separate link service acknowledgement message.

Under NoFlow, the link starts implicitly in a Send state. The transmitter is allowed to transmit until the receiver sends a link service message with a DontSend bit set. In this case, the receiver is responsible for keeping up with the transmitter. If the receiver is getting behind, it is then responsible for sending an "XOFF" message. If the receiver is poorly matched to the transmitter or if it has Ethernet hardware problems, excessive XON/XOFF messaging and retransmission will cause this situation to be worse than it would be under Segment Count.

However, if the protocol is request-response, the receive pipeline can be adjusted to receive all of a transmitted response and no flow control messages need be exchanged at all. NoFlow is the default type used by DECnet-VAX.

Additionally, if the sending and receiving systems have well-matched performance (that is, data can be processed as fast as it is being sent), the NoFlow option allows data to be exchanged with lower overhead than the Segment Count option.

- **DSO_LINKHOLD** allows the user to specify a time period in seconds during which the data retry algorithms continue to attempt to recover the link. This is in addition to the usual NSP retries, which are primarily controlled by the EXECUTOR RETRANSMIT FACTOR parameter. After this period, the link is disconnected with the message ENODEUNREACH. Use this value to override the global value set by the Network Control Program (NCP) for the EXECUTOR confidence timer. This value can range from 1 to 32767.

The default for this option is specified by the EXECUTOR CONFIDENCE TIMER parameter (currently 15 seconds). This timeout does not affect either link loss due to local resource failure or operation of the outgoing or incoming connection timers.

Input Arguments

<i>s</i>	specifies the number for a socket returned by the <i>socket</i> or the <i>accept</i> call.
<i>level</i>	specifies the level at which options are manipulated. The level is either SOL_SOCKET or DNPROTO_NSP. (See Appendix A for details.)
<i>optname</i>	specifies options to be interpreted at the level specified. For example, SO_LINGER at the SOL_SOCKET level.
<i>optval</i> , <i>optlen</i>	specify access option values. The interpretation of each argument is function dependent as shown here:
<i>optval</i>	specifies the address for a buffer which contains information for setting option values.
<i>optlen</i>	specifies the size of the option value buffer.

Return Values

If the call completes successfully, a value of 0 is returned. An unsuccessful call returns a value of -1. When an error condition exists, the external variable *errno* will contain error details. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[EACCES]	Socket in wrong state to change FLOWCTRL.
[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[ECONNABORTED]	The accept connect did not complete. The peer task disconnected and the connection was aborted.
[EDOM]	The acceptance mode is not valid.
[EFAULT]	The network received an invalid buffer.

setsockopt

[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOBUFS]	There are no available buffers for optional access control and/or user data.
[ENOPROTOOPT]	No access control information was supplied with the connection request.
[EOPNOTSUPP]	The option is unknown.

shutdown

NAME

shutdown – shutdown all or part of a full duplex logical link.

SYNTAX

```
int shutdown(s, how)
int s, how;
```

DESCRIPTION

The *shutdown* call causes all or part of a full duplex connection on the original socket to be shut down.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

how specifies the type of shutdown. The *how* argument can be set to:

- 0 which disallows further receives or reads.
- 1 which disallows further sends or writes.
- 2 which disallows further sends (or writes) and receives (or reads).

Return Value

If the *shutdown* call completes successfully, a value of 0 is returned. If an error occurs, a value of -1 is returned. Additional error detail will be contained in the external variable *errno*. See the **DIAGNOSTICS** section for a full description of the error messages.

shutdown

DIAGNOSTICS

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The specified socket is not connected.

ioctl

NAME

ioctl – control the operations of sockets.

SYNTAX

#Include <ioctl.h>

```

Int      ioctl(s, request, argp)
Int      s, request;
char     *argp; (or int *argp)
    
```

DESCRIPTION

The *ioctl* call controls the operations of sockets. The call indicates whether an argument is an input or output argument and the size of the specific argument in bytes.

NOTE

The DECnet-DOS function call *ioctl* is not compatible with DECnet-ULTRIX systems. See Appendix H for information on how to transport DECnet-DOS programs that use *ioctl*.

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

request specifies the I/O control function to be used. The control levels are:

FIONREAD returns the total byte count of all messages waiting to be read. The argument *argp* points to an int.

FIONBIO sets or clears blocking or nonblocking I/O operation. The argument *argp* points to a byte that contains a value of 0 or 1. For blocking I/O, *argp* should point to a value 0. For nonblocking I/O, *argp* should point to a value of 1.

sioctl

FIORENUM rennumbers an assigned socket number to another number. In this way, the original socket number is made available again. The valid range for socket numbers is 0 to 31. The argument *argp* points to an int.

NOTE

The *select()* function call cannot accept socket numbers that exceed this range. (See *select* for details.) If you specify a number that is already in use, an error message, **EEXIST**, is returned.

argp specifies the address of the argument list.

Output Argument

argp specifies the results of the socket operations.

Return Value

If the call completes successfully, a value of 0 is returned with the following additional message:

For **FIONREAD**, *argp* returns the total byte count of all messages waiting to be read.

If an error occurs, a value of -1 is returned. Additional error detail will be contained in the external variable *errno*. See the **DIAGNOSTICS** section for a full description of the error messages.

DIAGNOSTICS

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EEXIST]	The socket number is already in use.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[EOPNOTSUPP]	The socket type does not support the socket I/O operation.

socket

NAME

`socket` – create an endpoint for communication and return a socket number.

SYNTAX

```
#include <types.h>
#include <socket.h>
#include <dn.h>
```

```
int socket(domain, type, protocol)
int domain, type, protocol;
```

DESCRIPTION

The *socket* call creates a socket and returns a socket number. A socket is an addressable endpoint of communications within a task. It can be used to transfer data to or from a similar socket in another task. Subsequent function calls on this socket will refer to the associated socket number.

Input Arguments

domain specifies the communications environment as AF_DECnet.

type specifies the type of communication for the socket. For example, SOCK_STREAM. (See Appendix A for a list of defined socket types.)

SOCK_STREAM causes bytes to accumulate until internal DECnet buffers are full. The receiving task does not know how many bytes were sent in each write operation.

NOTE

To prevent a program from hanging on a stream socket, issue the *sioctl* call with the FIONREAD function argument (see *sioctl*), and then read only those numbers of bytes returned by the call. You should also perform socket operations in nonblocking I/O mode.

socket

`SOCK_SEQPACKET` causes bytes to be sent immediately. The receiving task receives those bytes in one "record".

protocol specifies a particular DECnet protocol to be used with the socket. (See Appendix A for a list of supported DECnet layers.)

Return Value

If the call completes successfully, the socket number is returned. This number is used by subsequent system calls on this socket. If an error occurs, a value of `-1` is returned. Additional error detail will be contained in the external variable *errno*. See the **DIAGNOSTICS** section for a full description of the error messages.

DIAGNOSTICS

[EAFNOSUPPORT]	The specified domain is not supported in this version of the system.
[EMFILE]	Too many open sockets.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.

sread

NAME

sread – read data from a socket.

DESCRIPTION

The *sread* call is used to read data from your peer. If no messages are available at the socket, the *sread* call waits for a message to arrive unless the socket is nonblocking. In this case, a status of `-1` is returned with the external variable *errno* set to `EWOULDBLOCK`.

If the socket becomes disconnected, queued data can still be received from the broken logical link. However, if you shut down the socket or detach it, queued data cannot be received. When the logical link is not in a connected state, and all data has been read, the *sread* call returns zero bytes.

The *select()* call can be used to determine if more data has arrived.

NOTE

The DECnet-DOS function call *sread* is not compatible with DECnet-ULTRIX systems. See Appendix H for information on how to transport DECnet-DOS programs that use *sread*.

The *sread* call performs the same function as the *recv* call with one exception – you cannot set any flags.

SYNTAX

int	sread(s, buffer, buflen)
int	s, buflen;
char	*buffer;

Input Arguments

s specifies the number for a socket returned by the *socket* or the *accept* call.

sread

buffer specifies the address of the buffer which will contain the received message.

buflen specifies the size of the message buffer.

Output Argument

buffer specifies the buffer which contains the received message.

Return Value

If the call succeeds, the number of read characters is returned.

If the call returns a zero, you have either received a zero length message or the logical link has been disconnected. To determine the state of the logical link, use the *getsockopt* function call with the *DSO_LINKINFO* option (see *getsockopt*), or the DECnet utility function, *dnet_eof* (see Chapter 5). If the link has been disconnected, then all subsequent receives will return zero bytes.

If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

When reading **normal data**, the following set of error messages can occur:

Blocking I/O Message

Description

[EBADF]

The argument *s* does not contain a valid socket number.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

**Nonblocking I/O
Message****Description****[EBADF]**

The argument *s* does not contain a valid socket number.

[ENETUNREACH]

The network is unreachable. The network process is not installed.

[EWOULDBLOCK]

The receive operation would block because there is currently no data to receive.

swrite

swrite

NAME

swrite – write data to a socket.

SYNTAX

```
Int      swrite(s, buffer, buflen)
Int      s, buflen;
char     *buffer;
```

DESCRIPTION

The *swrite* call is used to write data to your peer.

If no message space is available at the socket to hold the message to be transmitted, then the *swrite* call will normally block. If the socket has been placed in nonblocking I/O mode, the message will not be sent, and the function will complete with the error EWOULDBLOCK.

The *select()* call can be used to determine if more data may be sent over the socket.

NOTE

The DECnet-DOS function call *swrite* is not compatible with DECnet-ULTRIX systems. See Appendix H for information on how to transport DECnet-DOS programs that use *swrite*.

Input Arguments

<i>s</i>	specifies the number for a socket returned by the <i>socket</i> or the <i>accept</i> call.
<i>buffer</i>	specifies the address of the buffer which contains the outgoing message.
<i>buflen</i>	specifies the size of the message.

Return Value

If the call succeeds, the number of sent characters is returned. If an error occurs, a value of -1 is returned. Additional error detail will be specified in the external variable *errno*. See the **DIAGNOSTICS** section for a full description of the error messages.

DIAGNOSTICS

When sending **normal data**, the following set of error messages can occur:

**Blocking I/O
Message****Description**

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 4096 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ENOTCONN]	The <i>swrite</i> call did not complete and the link was disconnected.
[EPIPE]	The link has been disconnected, aborted, or shut down. No further messages can be sent.

**Nonblocking I/O
Message****Description**

[EBADF]	The argument <i>s</i> does not contain a valid socket number.
[EMSGSIZE]	The size of the outgoing message is more than 4096 bytes.
[ENETUNREACH]	The network is unreachable. The network process is not installed.

swrite

[ENOTCONN]

The *swrite* call did not complete and the link was disconnected.

[EPIPE]

The link has been disconnected, aborted, or shut down. No further messages can be sent.

[EWOULDBLOCK]

The outbound quota was full, and the message could not be sent.

C Language Subroutines

DECnet-DOSTM includes C language source files which are used to create a linkable library for DECnet-DOS applications. This library provides compatibility with the network socket interface supported by DECnet-ULTRIXTM.

The DECnetTM utility functions are contained in the C language source files. Some of these routines are used for accessing the network node database and manipulating the data.

Some of the DECnet utility functions include the DECnet header file <dnnetdb.h>. This file provides DECnet definitions used with standard C functions.

5.1 Creating the DECnet-DOS Programming Interface Library

The file DNETLIB.SRC contains three types of files: the .C files (C language sources), the .H files (header files that contain definitions for the network interface), and the .ASM files (assembly language sources). You should refer to the appropriate installation guide for a complete list of these files.

In order to interface with the DECnet-DOS network process, you need a library against which to link your DECnet-DOS program(s). If you installed DECnet-DOS from floppy diskettes, this library has already been created for you.

If you installed DECnet-DOS from tape (TK50), you must use the following procedure to create a DECnet-DOS programming interface library:

1. The Break Source utility, BREAKSRC, allows you to break the source file, DNETLIB.SRC, into separate source files for compilation and assemblies. BREAKSRC is supplied with the DECnet-DOS distribution kit. If you run the DECnet-DOS Installation Procedure (DIP), you can select to have the DNETLIB.SRC file split into separate files. The BREAKSRC utility will then be run automatically for you. For instructions on how to run DIP, refer to the appropriate installation guide.

To run BREAKSRC, use the following format:

```
BREAKSRC <input-file-spec> <output-device:path>
```

For example:

```
BREAKSRC A:DNETLIB.SRC C:\DECNET\SRC\
```

2. Use your C language compiler to compile each C language source module. Use your assembler to assemble each assembly source module.

NOTE

If you are using a Microsoft[®] C compiler, you must use the /zp switch when compiling your program.

3. After you produce an object module for each source module, build a library against which to link your DECnet-DOS applications programs.

5.1.1 DECnet-DOS Programming Considerations

The following programming considerations should be noted when writing and developing your DECnet-DOS applications:

1. Using the external variable *errno* — Most DECnet-DOS programming interface functions use the external variable *errno* as a place to return error detail. It is assumed that *errno* has been defined externally to the programming interface as an *int*. It may already be defined in your C language run-time library; if not, your applications program should define it.
2. Checking software compatibility — When creating DECnet-DOS applications, make sure that you resolve any C language compiler incompatibilities (such as long variable names or certain type definitions) before compiling the C language source modules.

3. If your C compiler does not do so by default, you should compile the sources so that all data structures are stored without extra space (packed) for alignment of members on "int" boundaries.
4. Using assembly source modules — There are assembly source modules included in DNETLIB.SRC. Before you can successfully call these functions from sources compiled by your C language compiler, you should fulfill any assembly-format requirements (such as segment names). (Refer to the header file, *begin.h*, on the distribution kit as an example of specific C language compiler segment naming requirements.)
5. Using specific macros — There are references to the macros/functions such as *toupper* and *islower* in some of the C language source modules. It is assumed that your C language compiler has provided a standard macro/function for them. If not, you can simply provide your own macros/functions.
6. If you are not using a C compiler that supports the *signal* handling function (for example, CTRL/C trapping), then you will need either to provide your own *signal* function or to comment out from the code in *dnet_conn* any references to the *signal()* function and the reference to the include file, *<signal.h>*.

The CTRL/C trapping, provided in *dnet_conn*, allows you to abort a utility which appears to be waiting indefinitely for connections to complete. (This applies only to utilities that use *dnet_conn* for making connections.) The *connect* function in *dnet_conn* is invoked in a nonblocking socket mode.

If CTRL/C trapping code is commented out from *dnet_conn*, there is the potential problem of leaving hung sockets and running out of system resources, should users decide to enter CTRL/C in the middle of nonblocking connection request attempts.

7. If you are using a compiler that does not define the following variables: *int daylight*, *long timezone* and *char *tzname[2]*, either comment these external declarations out of the header file *<time.h>* or define them in your C program code.

8. Using specific DECnet function calls — If you develop code that uses *dnet_getacc*, *dnet_installed*, and *dnet_path* function calls, you will be unable to transport that code to a DECnet-ULTRIX system. These function calls are not valid on DECnet-ULTRIX systems.
9. The DECnet-DOS programming interface library was based on a 2-segment model: one code segment and one data segment. Three sizes of models are supported: small model, medium model, and large model.

CAUTION

If a program terminates with any active logical links (sockets), the links remain active. In this way, another program can start and use the same links.

If the logical links are not needed, you must issue the *sclose* function call before a program is terminated with an `EXIT`, `CTRL/Z` or `CTRL/C`. If too many links are left active, an error message, indicating that no more buffers are available or that there are insufficient network resources, is reported. This causes the network to become unusable. To completely deactivate any logical links, run the Network Control Program (NCP) and issue the `SET KNOWN LINKS` command with `STATE OFF`. (See the *DECnet-DOS Network Management Guide* for more information on this command.)

5.2 DECnet Utility Function Calls

The following sections describe the DECnet utility function calls for C programs. The calls are summarized in Table 5-1.

Table 5-1: DECnet Utility Function Calls

DECnet Call	Description
<code>bcmp</code>	Compare byte strings.
<code>bcopy</code>	Copy <i>n</i> bytes from one specific string to another string.
<code>bzero</code>	Zero <i>n</i> bytes in a specific string.
<code>dnet_addr</code>	Convert an ASCII node address string to binary and return a pointer to a <i>dn_naddr</i> data structure.
<code>dnet_conn</code>	Connect to the specified target network object on a remote node and send along access control information and/or optional data.
<code>dnet_eof</code>	Test the current state of the connection.
<code>dnet_getacc</code>	Search the incoming access database file, DECACC.DAT, for access control information that is associated with a given user name. The access control information set by the NCP command SET ACCESS is stored in the database file, DECACC.DAT.
<code>dnet_getalias</code>	Return default access control information by node name.
<code>dnet_htoa</code>	Search the node database. If the node name is found, a pointer to the DECnet ASCII node name string is returned. Otherwise, a pointer to the DECnet ASCII node address string is returned. If the function fails to return a valid node name or address string, a pointer to the string "?unknown?" is returned.
<code>dnet_installed</code>	Perform an installation check on the specific software module.
<code>dnet_ntoa</code>	Specify a pointer to the <i>dn_naddr</i> data structure which contains the binary node address. If the function completes successfully, a pointer to the ASCII string representation of the DECnet node address is returned.
<code>dnet_path</code>	Return a modified file name which contains the DECnet data base device and path name prefixed to the file name.
<code>getnodeadd</code>	Return the address of your local DECnet-DOS node.
<code>getnodeent</code>	Access the network node database and return complete node information given only a node address or node name.
<code>getnodename</code>	Return the ASCII string representation of your local DECnet-DOS node.
<code>nerror</code>	Produce DECnet error messages and output the ASCII text string to <i>stdout</i> .
<code>perror</code>	Produce an ULTRIX error message appropriate to the last detected system error, and output the ASCII text string to <i>stdout</i> .

bcmp

bcmp

NAME

bcmp – compare byte strings.

SYNTAX

Int **bcmp**(**s1**, **s2**, **n**)

char ***s1**, ***s2**;
Int **n**;

DESCRIPTION

bcmp compares byte strings to see if they are matching character strings. It is assumed that the strings are of equal length.

Input Arguments

s1 specifies the address of the first character string.
s2 specifies the address of the second character string.
n is the length of the strings.

Return Value

If a match is found, the value of 0 is returned. Otherwise, a nonzero value is returned.

bcopy

NAME

bcopy – copy *n* bytes from one specific string to another string.

SYNTAX

```
int bcopy(s1, s2, n)
```

```
char    *s1,*s2;  
int     n;
```

DESCRIPTION

bcopy copies *n* bytes from one specific string to another string.

Input Arguments

s1 is the character pointer to the source string.

s2 is the character pointer to the destination string.

n specifies the number of bytes to be copied.

Return Value

The number of bytes copied from the source string to the destination string is returned.

bzero

bzero

NAME

bzero – zeroes *n* bytes in a specified string.

SYNTAX

int bzero(s1, n)

char *s1;
int n;

DESCRIPTION

bzero zeroes *n* bytes in a specified string.

Input Arguments

**s1* is the character pointer to the specified string.

n specifies the number of bytes to be zeroed.

Return Value

The number of bytes zeroed in the specified string is returned.

dnet_addr

NAME

dnet_addr – convert an ASCII node address string to binary and return a pointer to a *dn_naddr* data structure.

SYNTAX

```
struct dn_naddr *dnet_addr(cp)
```

```
char *cp;
```

DESCRIPTION

In area based networks, a DECnet node address includes an area number and a node number. The function call *dnet_addr* converts an ASCII node address string to binary and returns a pointer to a *dn_naddr* data structure. This information is required for the *sockaddr_dn* data structure. (See Appendix B on how these data structures are formatted.)

Input Argument

cp is the character pointer to the ASCII node address string. The DECnet node address is specified as *a.n*

where

a is the area number

n is the node number

dnet_addr

Return Argument

dn_naddr specifies the node address data structure. A user retrieves data from the fields filled in by this function call. The fields are:

a_len specifies the length of the returned node address.

a_add specifies the node address.

If the call succeeds, a pointer to a *dn_naddr* data structure is returned. Otherwise, a null value is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local buffer.

dnet_conn

NAME

dnet_conn—connect to the specified target network object on a remote node and send along access control and/or optional user data.

SYNTAX

```
int dnet_conn(node, object, sock_type, out_data, out_len,  
in_data, in_len)
```

```
char *node; char *object; int sock_type; u_char *out_data,  
*in_data, int out_len, *in_len;
```

DESCRIPTION

dnet_conn establishes a connection to the specified target DECnet object on a remote node. If no access control information is supplied as part of the node input argument, the default access control information (if found in the access control database) will be sent. Optional data can also be passed with the function.

dnet_conn supports password prompting based on the input node specification string. You are asked to supply a password whenever:

- The password field in the node specification is either a question mark (?) or an asterisk (*).
- The user field is present but the password field is missing.

The following example node specifications will cause prompting for passwords:

```
dnet_conn("boston/revere",...)  
dnet_conn("boston/revere/?",...)  
dnet_conn("boston/revere/*",...)
```

dnet_conn

dnet_conn supports outgoing proxy log-in access. Outgoing proxy allows the local node to initiate proxy log-in access to the remote node, but does not allow proxy log-in access from the remote node to the local node. Before you are permitted to use proxy log-in, the following must take place:

- Proxy log-in access **must** be supported at the remote node.
- The Executor (local) node must have a user name set up in its node database. This user name is passed in outgoing connection requests and may be used for proxy log-in access. To set up access control information, refer to the discussion of the NCP utility in the *DECnet-DOS Network Management Guide*.

If access control information is not explicitly supplied with an input node name, *dnet_conn* will check the node database for implicit access control information. If implicit access control does not exist, proxy log-in will take place with the Executor node's user name and passed in the outgoing connection request.

The following examples show the use of proxy log-in access with the *dnet_conn* function call:

The Executor (local) node has set up TASHA as the user name in the local node's database. The remote node BOSTON is stored as an entry in the local node's database without any access control information.

1. No proxy, null access control information is explicitly specified:

```
dnet_conn("BOSTON//",...)
```

2. Proxy will be passed, no explicit access control information, no implicit access control information for node BOSTON in the database:

```
dnet_conn("BOSTON",...)
```


The Executor (local) node has set up TASHA as the user name. The remote node BOSTON is stored as an entry in the local node's database with TASHA as the user name/access control information.

1. No proxy, null access control information is explicitly specified:

```
dnet_conn("BOSTON//", ...)
```

2. No proxy, the implicit access control information for node BOSTON will be used:

```
dnet_conn("BOSTON", ...)
```

A target task can return a 1- to 16-byte optional data message when it accepts or rejects the connection request.

When a program which uses *dnet_conn* fails to complete a connection request, *nerror* can subsequently be called in order to display the DECnet error message.

Input Arguments

node specifies the address of the string which contains the remote node name or address and any access control information. Node names are always converted to uppercase before being processed by this function. Access control information is passed as supplied with regard to case.

To pass access control information, the node string can take one of the following formats:

```
"node-name/user/password/account"
```

or

```
"area.node-number/user/password/account"
```

dnet_conn

To pass null access control information, the node string can take one of the following formats:

"node-name/1"

or

"area.node-number/1"

object specifies the address of the string which contains the specified target DECnet object. The object string can be specified in one of the following ways:

1. To access a target object by supplying an object name. You should note that the object name is passed as supplied, with regard to case.

"target-object-name"

2. To access a target object by object number.

"#target-object-number"

sock_type is 0 when creating a sequenced socket packet and 1 when creating a stream socket.

out_data specifies the address of the outgoing optional user data buffer. If not required, supply a null pointer.

out_len specifies the size of the optional outgoing message. The message length can be up to 16 bytes. If not required, supply a null value.

in_data specifies the address of the buffer which will store the optional incoming message. If not required, supply a null pointer.

in_len specifies the address of the location in which the value will be stored. It should initially contain the size of the buffer pointed to by *in_data*. On return, it will contain the actual size of the optional incoming message. If not required, supply a null pointer.

Return Value

If the function completes successfully, the socket number is returned. If an error occurs, a value of -1 is returned. Additional error detail will be contained in the external variable *errno*. See the DIAGNOSTICS section for a full description of the error messages.

DIAGNOSTICS

[E2BIG]	An argument is too long.
[EACCES]	Access control information was rejected.
[EADDRNOTAVAIL]	The node name is undefined.
[EDESTADDRREQ]	A specific destination address is required.
[ENAMETOOLONG]	The node name is invalid.
[ENETUNREACH]	The network is unreachable. The network process is not installed.
[ESRCH]	The object is unknown.

NOTE

Additional errors may be returned by the *socket*, *setsockopt*, and the *connect* function calls which are called from within *dnet_conn*.

dnet_eof

dnet_eof

NAME

dnet_eof – test the current state of the connection.

SYNTAX

Int dnet_eof(sock)

Int sock;

DESCRIPTION

dnet_eof tests the current state of the connection.

Input Argument

sock specifies the socket whose connection is to be checked.

Return Value

If the connection is determined to be active (either a running or a connecting state), a value of 0 is returned. If the connection is inactive, a value of 1 is returned.

NOTE

If a bad socket number is supplied with *dnet_eof*, a value of 1 is returned. However, this value is not a true indication of an invalid socket number.

dnet_getacc

NAME

dnet_getacc – searches the incoming access database for access control information that is associated with a given user name.

SYNTAX

```
struct dnet_accnt *dnet_getacc(nacc)
struct dnet_accnt *nacc;
```

DESCRIPTION

dnet_getacc searches the incoming access database file, DECACC.DAT, for access control information that is associated with a given user name. The access control information set by the NCP command SET ACCESS is stored in the database file, DECACC.DAT.

NOTE

If you develop code that uses *dnet_getacc*, you will be unable to transport that code to a DECnet-ULTRIX system. This call is not valid on DECnet-ULTRIX systems.

Input Argument

nacc is a character pointer to an access control information block containing the user name to be matched. The string consists of 1 to 39 alphabetic characters. This string must be identical to the user string set up by the NCP command SET ACCESS. (Use the NCP command SHOW KNOWN ACCESS to display the string.) Refer to the *DECnet-DOS Network Management Guide* for information on using these commands.

Return Argument

nacc specifies the access control information block data structure *dnet_accnt*. A user retrieves information filled in by this function call. (See Appendix B on how *dnet_accnt* is formatted.)

dnet_getacc

The following data fields can be filled in by this function call:

<i>acc_status</i>	is used internally by this function call.
<i>acc_type</i>	specifies the type of privilege associated with a user name. The four access types are: 0 for no access rights, 1 for read only access, 2 for write only access, and 3 for read and write access.
<i>acc_user</i>	specifies the user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.
<i>acc_pass</i>	specifies the password associated with a user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.

If there is a match, a character pointer to the access control information block associated with the user string is returned. The access type is always returned along with any user and password information.

A value of 0 is returned if there is no match or if the DECnet database path cannot be found for DECACC.DAT using the function *dnet_path*.

NOTE

If you plan to use this function again before you are finished using the data, you must copy the data into a local structure.

dnet_getalias

NAME

`dnet_getalias` – returns default access control information by node name.

SYNTAX

```
char *dnet_getalias(node)
```

```
char *node;
```

DESCRIPTION

`dnet_getalias` retrieves any default access control information associated with a specific node.

Input Argument

node is a character pointer to a node name. The node name is forced to uppercase and then processed by the function.

Return Argument

If the node has default access control information associated with it, the node name followed by the access data is retrieved. The extended node name string is returned as *node-name/user/password/account*. This data was set up in the permanent databases, DECNODE.DAT and DECALIAS.DAT, using the Network Control Program (NCP).

If no access control data can be found, a null pointer is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local buffer.

dnet_htoa

dnet_htoa

NAME

`dnet_htoa` – search the node database. If the node name is found, a pointer to the DECnet ASCII node name string is returned. Otherwise, a pointer to the DECnet ASCII node address string is returned. If the function fails, a pointer to the string “?unknown?” is returned.

SYNTAX

```
char *dnet_htoa(add)
```

```
struct dn_naddr *add;
```

DESCRIPTION

`dnet_htoa` searches the node database. If the node name is found, a pointer to the DECnet ASCII node name string is returned. If the node name is not found, a pointer to the DECnet ASCII node address string is returned.

Input Argument

add specifies a pointer to a structure of the type `dn_naddr`, which contains the node address. The format is *area.number*. (Refer to Appendix B for information on how the `dn_naddr` data structure is formatted.)

Return Value

If the node name is found, a pointer to the DECnet ASCII node name string is returned. Otherwise, a pointer to the DECnet ASCII node address string is returned.

If the function call fails to return a valid node name or address string, a pointer to the string “?unknown?” is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local structure.

dnet_installed

NAME

dnet_installed – perform an installation check on a specific software module.

SYNTAX

```
Int dnet_Installed(vector, tla)
short vector;
char *tla;
```

DESCRIPTION

dnet_installed performs an installation check on a specific software module. You supply the interrupt vector number and the 3-letter acronym for the software module.

If the software module is installed, the 2-byte software version number is returned. The low byte is the major version number. The high byte is the minor version number. Otherwise, a value of -1 is returned.

NOTE

If you develop code that uses *dnet_installed*, you will be unable to transport that code to a DECnet-ULTRIX system. This function call is not valid on DECnet-ULTRIX systems.

Input Arguments

<i>vector</i>	specifies the interrupt vector number for the software module. Appendix A lists the interrupt vector numbers for the defined software modules.
<i>tla</i>	is the 3-letter acronym for the software module. The acronym is passed as supplied, with regard to case. For example, DNP is the acronym for the DECnet Network Process. It must be passed as uppercase. See Appendix A for a list of defined software modules.

dnet_installed

Return Value

If the installation check successfully completes, the 2-byte software version number is returned. Otherwise, a value of -1 is returned.

dnet_ntoa

NAME

`dnet_ntoa` – convert a DECnet node address from binary form to ASCII form.

SYNTAX

```
char *dnet_ntoa(add)
```

```
struct dn_naddr *add;
```

DESCRIPTION

dnet_ntoa converts a DECnet node address from binary form to ASCII form.

Input Argument

add specifies a pointer to a structure of the type *dn_naddr*, which contains the binary node address. (See Appendix B on how the *dn_naddr* data structure is formatted.)

Return Value

If the function completes successfully, a pointer to the ASCII string representation of the DECnet node address is returned. The format is *area.number*.

If the function call fails to return a valid node name or address string, a pointer to the string “?unknown?” is returned.

NOTE

If you plan to call this function again before you are finished using the data, you must copy the data into a local structure.

dnet_otoa

dnet_otoa

NAME

dnet_otoa — convert DECnet object name or number to an ASCII string.

SYNTAX

```
char      *dnet_otoa(dn)
struct    sockaddr_dn dn;
```

DESCRIPTION

Given a `sockaddr_dn` data structure, `dnet_otoa` converts a DECnet object name or number to its ASCII representation.

Input Argument

dn is the address of a structure `sockaddr_dn`. Refer to Appendix B for information on how the `sockaddr_dn` structure is formatted.

Return Value

A character pointer to the ASCII string representation of the DECnet object name or number is returned.

NOTE

If you plan to call this function again before you finish using the data, you must copy the data into a local structure.

dnet_path

NAME

`dnet_path` – return a modified file name that contains the DECnet database device and path name prefixed to the specified input file name.

SYNTAX

```
char *dnet_path(file_name)
```

```
char *file_name;
```

DESCRIPTION

Given a character string pointer to a file name, *dnet_path* returns a modified file name that contains the DECnet database device and path name prefixed to the specified input file. It is recommended that all user-created DECnet-DOS database files be located in the same DECnet directory.

For example:

Call:

```
new_file_name = dnet_path("NCPHELP.BIN");
```

Returns:

```
new_file_name = "c:\decnet\NCPHELP.BIN"
```

The DECnet database device and path name should be specified as input arguments to the DLL and/or DNP command lines in your AUTOEXEC.BAT file. To do this, edit your AUTOEXEC.BAT file using EDLIN or a similar text editor.

dnet_path

Important

The *dnet_path* function call differs from Version 1.0. You can no longer set the DECnet database device and path names using the NCP command SET EXECUTOR. However, you can still display the database path specification. To do this, issue the NCP command SHOW EXECUTOR CHARACTERISTICS. For information on using this command, refer to the *DECnet-DOS Network Management Guide*.

If you develop code that uses *dnet_path*, you will be unable to transport that code to a DECnet-ULTRIX system. This function call is not valid on DECnet-ULTRIX systems.

When the system is rebooted, DNP and/or DLL will use its command line argument or default to the current device:\decnet as the path specification.

To change the DECnet database path, you will need to specify a new path specification as command line input the next time that DLL and/or DNP is installed.

DECnet-DOS Version 2.0 supports Ethernet and asynchronous DDCMP configurations. The following examples illustrate acceptable ways for specifying the DECnet database path.

Examples

If you have an Ethernet setup

1. The Scheduler (SCH), Data Link layer (DLL), and DECnet Network Process (DNP) files are to be installed. The DLL and DNP command lines include a defined DECnet database path. The DLL path is used.

```
.  
. .  
. .  
SCH  
DLL c:\decnet\v11  
DNP c:\decnet\v11
```

2. SCH, DLL, and DNP files are to be installed. Only the DLL command line includes a defined DECnet database path. A database path is not defined for DNP. The database path set for DLL will also be used by DNP.

```
.
.
.
SCH
DLL c:\decnet\v11
DNP
```

3. SCH, DLL, and DNP files are to be installed. The DLL and DNP command lines do not include defined database paths. For this setup, the default DECnet database path (current device:\decnet) set for DLL will also be used by DNP.

```
.
.
.
SCH
DLL
DNP
```

If you have an asynchronous DDCMP setup

1. SCH and DNPDCP files are to be installed. The DNPDCP command line includes a defined DECnet database path. The DLL file is not required for asynchronous operations.

```
.
.
.
SCH
DNPDCP c:\decnet\v11
```


dnet_path

2. SCH, and DNPDCP files are to be installed. A DECnet database path is not specified for DNPDCP. Therefore, the default database path (current drive:\decnet) will be used by DNPDCP .

·
·
·

SCH
DNPDCP

Return Value

If the call completes successfully, a pointer to a modified file name that contains the DECnet database device and path name prefixed to the file is returned.

getnodeadd

NAME

getnodeadd – return the address of your local DECnet-DOS node.

SYNTAX

```
struct dn_naddr *getnodeadd();
```

DESCRIPTION

getnodeadd returns a pointer to a *dn_naddr* data structure which contains the DECnet node address of the local DECnet-DOS node.

Return Argument

dn_naddr specifies the node address data structure. A user retrieves data from the fields filled in by this function call. The fields are:

a_len specifies the length of the returned DECnet-DOS node address.

a_add specifies the DECnet-DOS node address.

If the call succeeds, a pointer to a *dn_naddr* data structure is returned. If an error occurs, a null value is returned.

NOTE

The *dn_naddr* data structure will be reused by additional calls. To keep this information, you must copy the data into a local structure.

getnodeent

getnodeent

NAME

`getnodeent`, `getnodebyaddr`, `getnodebyname`, `setnodeent`, `endnodeent` – access the network node database and return complete node information given only a node address or node name.

SYNTAX

```
#include <dnetdb.h>
```

```
struct    nodeent *getnodeent()
```

```
struct    nodeent *getnodebyname(name)
char      *name;
```

```
struct    nodeent *getnodebyaddr(addr, len, type)
char      *addr;
int       len, type;
```

```
setnodeent(stayopen)
int       stayopen;
```

```
endnodeent()
```

DESCRIPTION

These functions access the network node database and return complete node information given only a node address or node name. Each *getnodeent*, *getnodebyname* and *getnodebyaddr* function returns a pointer to a single static *nodeent* structure. This structure contains the broken out fields of an entry in the network node database.

getnodeent returns a pointer to the next entry of the database.

setnodeent positions you at the beginning of the database. If the *stayopen* flag is set to nonzero, the node database is not closed after each call to *getnodeent* (either directly, or indirectly through one of the other “getnode” calls).

endnodeent closes the database file.

getnodebyname and *getnodebyaddr* sequentially search from the beginning of the database until a matching node name or node address is found or until the end of the database is encountered. Node names are stored in the network node database as uppercase. Therefore, comparisons of node name strings to node names stored in the database are forced to be uppercase. Node addresses are always arranged in ascending numeric order.

Input Arguments

<i>name</i>	specifies the address of the buffer containing the DECnet node name string.
<i>addr</i>	specifies the address of the buffer containing the DECnet node address string.
<i>len</i>	is the length of the node's address string in bytes.
<i>type</i>	is the address type AF_DECnet.
<i>stayopen</i>	specifies a call dependent argument. If set to nonzero, the network node database is kept open for subsequent "getnode" calls.

Return Value

If the function, other than *setnodeent* and *endnodeent*, completes successfully, the address for the *nodeent* structure is returned.

<i>nodeent</i>	specifies the node address database structure. A user retrieves data from the fields filled in by this function call. The following data fields can be modified:
<i>*n_name</i>	points to a string which is the name of the DECnet node.
<i>n_addrtype</i>	specifies the address type AF_DECnet.
<i>n_addr</i>	specifies the DECnet network address for the node.

getnodeent

If an error or an EOF occurs, a null pointer is returned.

If *setnodeent* completes successfully, a value of 0 is returned. Otherwise, a value of -1 is returned.

NOTE

The *nodeent* structure will be reused by additional calls. To keep this information, you must copy the data into a local structure.

getnodename

NAME

getnodename – return the DECnet node name of your local DECnet-DOS node.

SYNTAX

```
char *getnodename();
```

DESCRIPTION

getnodename returns the ASCII string representation of your local DECnet-DOS node name.

Return Value

If the function call is successful, your local DECnet node name is returned. Otherwise, a null pointer is returned.

nerror

nerror

NAME

nerror – produce DECnet system error messages.

SYNTAX

```
nerror(s)
char *s;
```

DESCRIPTION

nerror produces DECnet error messages by mapping standard *errno* values to their equivalent DECnet error messages. First the characters pointed to by *s* are output to *stdout*, followed by a colon, and then the resulting DECnet error text. The error number is taken from the external variable, *errno*, which is set when an error occurs.

If a program makes a call to *dnet_conn* which fails, *nerror* should be subsequently called in order to display the DECnet error text.

Input Argument

**s* is the character pointer to the ASCII text string to be displayed before the DECnet error text is displayed.

Output Argument

The characters pointed to by *s* are output to *stdout*, followed by the colon, and then the resulting DECnet error text. You should refer to the section on *dnet_conn* for the list of DECnet error messages returned by *dnet_conn*.

NOTE

This call is not ULTRIX compatible. For the ULTRIX call, the log is output to *stderr*.

perror

NAME

perror – produce a standard ULTRIX error message appropriate to the last detected system error.

SYNTAX

perror(*cp*)

char **cp*;

DESCRIPTION

perror produces a standard ULTRIX error message appropriate to the last detected system error. First the character string is output, followed by a colon, and then the standard ULTRIX error message indexed by *errno*.

For example:

```
perror("Last error was ")
```

Input Argument

**cp* is the character pointer to the ASCII text string to be displayed before the ULTRIX error text is displayed.

NOTE

The log from *perror* is output to *stdout*. This call is not ULTRIX compatible. For the ULTRIX call, the log is output to *stderr*.

1911

1. The first part of the report is devoted to a general description of the country and its resources.

2. The second part is devoted to a description of the principal industries and occupations.

3. The third part is devoted to a description of the principal cities and towns.

4. The fourth part is devoted to a description of the principal rivers and lakes.

5. The fifth part is devoted to a description of the principal mountains and hills.

6. The sixth part is devoted to a description of the principal forests and parks.

7. The seventh part is devoted to a description of the principal lakes and ponds.

8. The eighth part is devoted to a description of the principal rivers and streams.

9. The ninth part is devoted to a description of the principal mountains and hills.

10. The tenth part is devoted to a description of the principal forests and parks.

11. The eleventh part is devoted to a description of the principal lakes and ponds.

12. The twelfth part is devoted to a description of the principal rivers and streams.

13. The thirteenth part is devoted to a description of the principal mountains and hills.

14. The fourteenth part is devoted to a description of the principal forests and parks.

15. The fifteenth part is devoted to a description of the principal lakes and ponds.

16. The sixteenth part is devoted to a description of the principal rivers and streams.

17. The seventeenth part is devoted to a description of the principal mountains and hills.

18. The eighteenth part is devoted to a description of the principal forests and parks.

19. The nineteenth part is devoted to a description of the principal lakes and ponds.

20. The twentieth part is devoted to a description of the principal rivers and streams.

Assembly Language Task-to-Task Interface

DECnet-DOSTM implements a session-level socket interface that is functionally compatible with the 4.2BSD UNIX[®] network interface. The direct programming interface to the DECnet-DOS Network Process lets you implement programs that can use all of the functions of the DECnetTM session layer. These programs make use of DECnet's full task-to-task capabilities to communicate with programs that reside on different DECnet network nodes.

This chapter describes the direct programming interface to the DECnet-DOS network process.

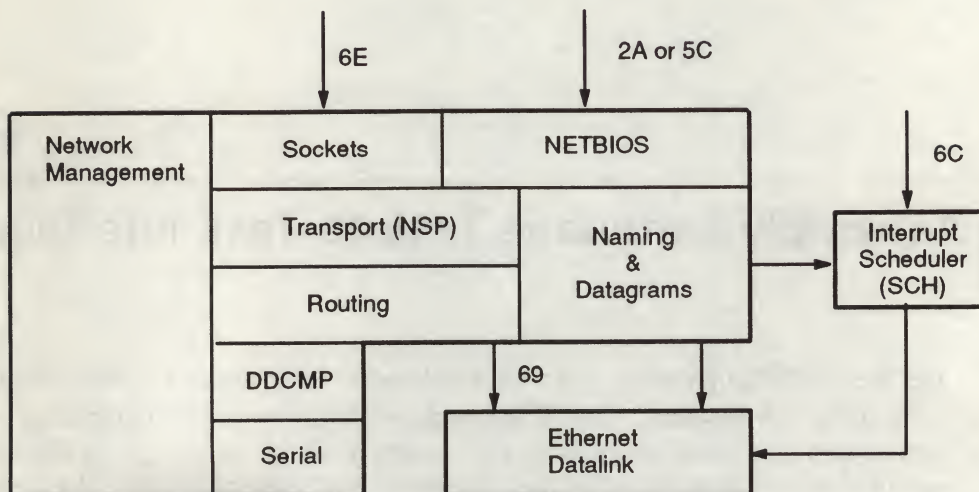
6.1 Introduction

By accessing the DECnet-DOS Network Process (DNP), an application that you write can communicate with another program on a remote DECnet node. Your application accesses the DNP by issuing function calls to:

- Initiate, accept, and establish a communications link
- Send and receive data
- Terminate a communications link

The DNP is an MS-DOS[®] terminate and stay resident task named DNPEXE. You can invoke DNP from a batch file or by typing DNP at the DOS prompt.

The following diagram illustrates DNP and its support modules:



6.1.1 How Applications Communicate

The socket interface is based upon a client-server model that prescribes the conditions under which two programs in a network can communicate. This section provides a brief review of client-server concepts. For more detailed information, refer to Chapter 1 of this manual .

When writing programs that implement full task-to-task communication, you should remember that your program may be either a client program or a server program. That is, your program may either initiate a connection and request services from another program; or, your program may wait for a connection request and provide services for another program.

A server program may be either another user application or it may be a predefined DECnet object. Network objects are identified on the network by name and number as defined in the network object database. A client program uses this object number to specify a connect request to a network object. See Appendix A for a list of the DECnet objects to which your client program can connect. Appendix A lists the name and number of each DECnet object.

6.1.2 Establishing Sessions

DECnet-DOS client and server programs use sockets to establish a communications session (also referred to as a DECnet logical link). A socket is an addressable endpoint for communication between the two programs. A program uses the socket to send data to, and receive data from, a socket or its equivalent in another network program. The programs use the network process function calls to:

- Create a socket
- Assign a socket number
- Listen for incoming connections
- Request or accept connections
- Send and receive normal or out-of-band data
- Detach and disconnect sockets and logical links

The DECnet-DOS socket interface supports the following socket types:

- Stream sockets. A stream socket provides for the bidirectional, reliable, sequenced, and unduplicated flow of data without record boundaries.
- Sequenced packet sockets. A sequenced packet socket is the same as a stream socket except that it preserves record boundaries in data.

6.2 Accessing DECnet-DOS Network Process Services

Your application program sends data to the DECnet-DOS Network Process (DNP) by issuing a series of function calls. These DNP function calls provide the information needed to establish a communications session with a target application on a remote node.

To issue function calls to the DNP, your application must build a special network data structure for each call, then issue a DOS software interrupt to vector 6EH (INT 6E). DNP uses this data structure, the Network I/O Control Block (NIOCB), to exchange information with your application program.

6.2.1 DECnet Network Process Installation Check

Before you attempt to access the DECnet network process, your application should check to see if the network process has been installed. Do this by issuing the MS-DOS Get Interrupt Vector function, with the DNP vector number 6EH as input. DOS will return a pointer to the DNP interrupt entry point.

The following illustrates the installation check:

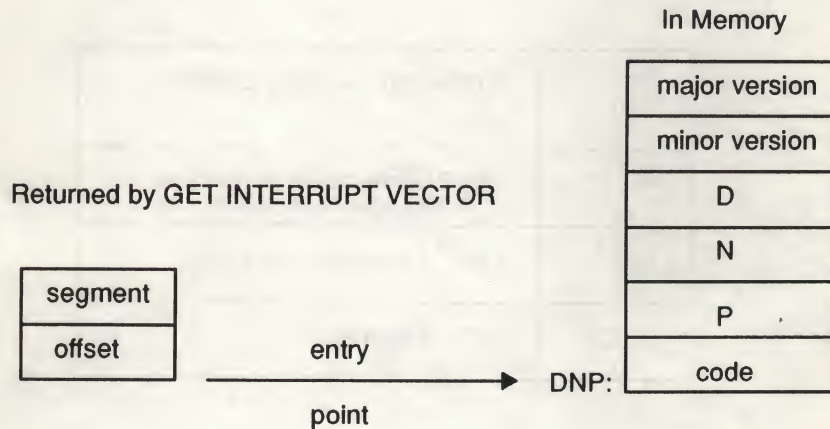
On Entry	8086/8088 Register Contents
AH	35H (hexadecimal function code)
AL	6E (interrupt vector number for the DECnet network process)

On Return	8086/8088 Register Contents
ES:BX	Pointer to the interrupt handling routine

On entry, the AH register must contain the hexadecimal code, 35H. The AL register must contain the interrupt number for the DECnet network process. Then the application must call MS-DOS through INT 21H. On return, the ES:BX register contains the address of the DNP entry point.

You must examine the contents of the five bytes that precede the DNP entry point to determine if the DNP module is installed, and if so, to obtain the software version number for this module. Of these five bytes, the first three bytes constitute a 3-letter name for the DNP software module. The next two bytes constitute the software version number: the low byte is the major version, the high byte is the minor version.

For example:



6.2.2 Calling the Network Process

To issue function calls to the DNP, your application must build an NIOCB data structure for each call, then issue a software interrupt to vector 6EH (INT 6E). Your application must complete the following four-step process to present calls to the DNP.

1. Create a Network I/O Control Block (NIOCB) by:
 - Building and filling required fields
 - Allocating any necessary buffers
 - Leaving 20 bytes of stack space for each outstanding NIOCB call
2. Set a pointer to the NIOCB address

3. Set the function code in the registers AH and AL
4. Issue the software interrupt to vector 6E hex (INT 6EH)

For each call your application issues, the DNP returns status codes to your application in special status and error fields of the NIOCB. These fields and messages are described later in this chapter.

The following illustrates the 6EH function call:

On Entry	8086/8088 Register Contents
AH	DE (DECnet network process hexadecimal code)
AL	1 (NIOCB function request)
DS:DX	NIOCB address

6.3 The Network I/O Control Block (NIOCB)

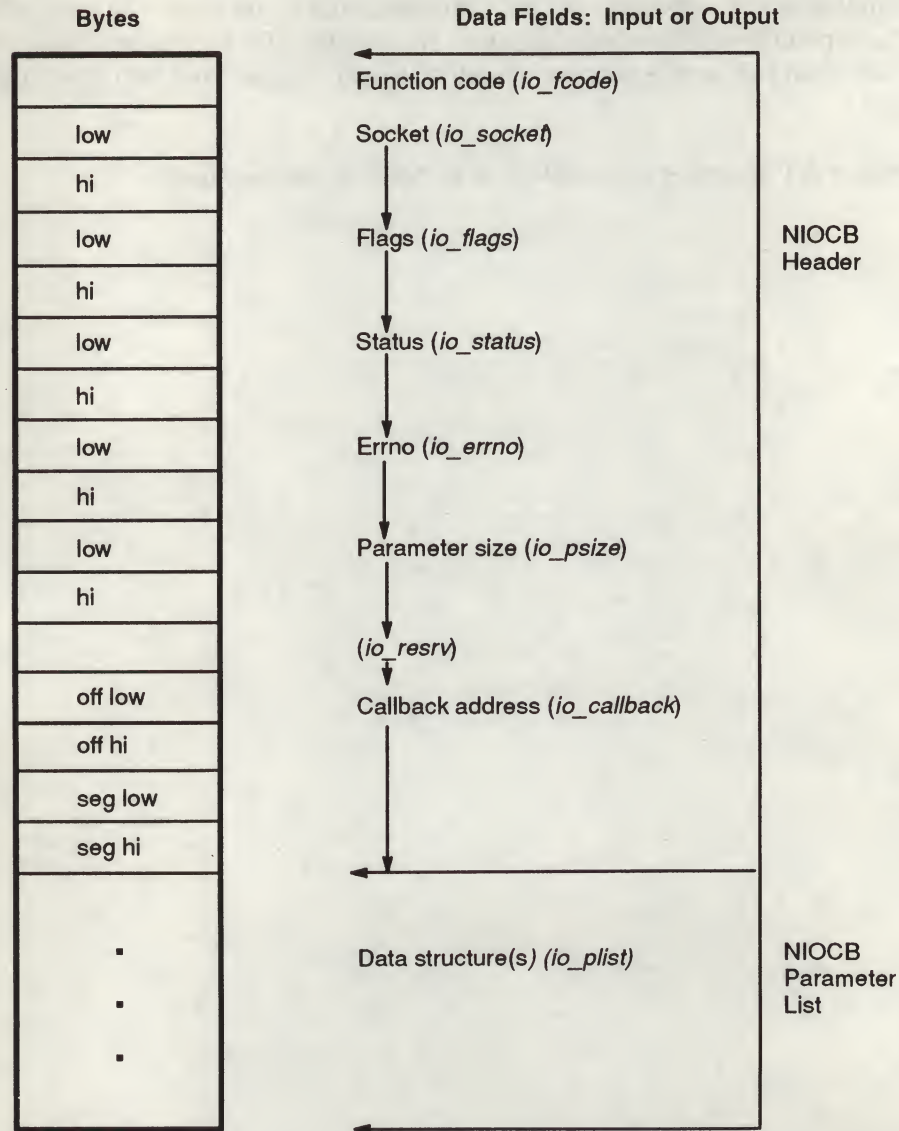
You use the DNP function calls to communicate with the network process. However, before your application program can issue a network process call, the program must construct a Network I/O Control Block (NIOCB) data structure. The DNP uses the NIOCB to exchange information with your application program.

Your program supplies values for the various NIOCB fields, or uses the default values, for each DNP call. The NIOCB must contain all the information specific to the function you want to perform. After it builds an NIOCB, your program can issue a call to the network process, through INT 6EH, passing the address of the NIOCB to the DNP.

Each NIOCB consists of header fields and a parameter list. The header field sub-structure, which is fixed in size, is used in every call to DNP. The parameter list is variable in size, depending upon the function request you issue. The parameter list fields point to additional data structures you may need for various calls. Refer to the individual call descriptions for the NIOCB fields required for each DNP function call.

Figure 6-1 illustrates the members of an NIOCB data structure.

Figure 6-1: An NIOCB Data Structure



LKG-3086-891

Table 6-1 summarizes the NIOCB header and parameter fields. Refer to the following sections for detailed descriptions of these fields. Note that the structures described in this table are defined for you in the header files supplied with the DECnet-DOS kit.

Table 6-1: NIOCB Data Fields

NIOCB Field	Size (Bytes)	Description
Header Members		
<i>io_fcode</i>	1	Function code field. Use this field to pass the function code to the network process.
<i>io_socket</i>	2	Socket number field. This field specifies the value, which is returned by the DNP, for a socket. The network process will use this number to identify the socket on successive calls.
<i>io_flags</i>	2	Flag option field. Use this field to specify function options such as asynchronous I/O and callback routines.
<i>io_status</i>	2	Command status field. The DNP returns values to this field as follows to indicate status. 0 = call completed successfully -1 = error completing call -2 = call is pending
<i>io_errno</i>	2	Error code field. The DNP returns additional error codes to this field if an "error completing call" is returned to the <i>io_status</i> field.
<i>io_psize</i>	2	Parameter list size or buffer size.
<i>io_resv</i>	1	Reserved for future use. Must be zero.
<i>io_callback</i>	4	Callback address. This field specifies the far address of an optional completion routine.
Parameter List Members *		
<i>attach_dn</i>	12	Attach function data structure. Stores socket creation data.
<i>io_buffer</i>	4	Data buffer address.
<i>listen_dn</i>	2	Listen data structure. Defines maximum number of incoming connects.

Table 6-1 (Cont.): NIOCB Data Fields

NIOCB Field	Size (Bytes)	Description
<i>localinfo_dn</i>	20	Local node information data structure.
<i>select_dn</i>	16	Select data I/O descriptors.
<i>shutdown_dn</i>	2	Shut down logical link.
<i>sioctl_dn</i>	8	Socket I/O control.
<i>sockaddr_dn</i>	26	Socket definitions.
<i>sockopt_dn</i>	12	Socket options.
<i>buffer_dn</i>	6	Data buffer descriptor.

* See Appendix B for more information about these structures.

6.4 Programming Tips and Considerations

This section provides programming considerations that you should keep in mind when writing your network application.

6.4.1 Handling Network I/O

DECnet-DOS supports three ways of handling network I/O: blocking synchronous, nonblocking synchronous, and asynchronous I/O. You should use the mode that best suits your application. Each method varies from the others in the way your application program processes and receives status information from the network process. These differences are detailed in the following sections.

6.4.1.1 Blocking Synchronous Mode

If your application issues a call in blocking synchronous mode, your application stops processing, or waits, until the network process has completed the call and returns status to your application. Processing resumes upon completion.

The DNP returns status information to your application in the *io_status* and *io_errno* fields of the NIOCB data structure immediately after processing your call.

6.4.1.2 Nonblocking Synchronous Mode

If your application issues a call on a nonblocking socket, DNP will either process and complete the call immediately, or will return an indication (EWOULDBLOCK) that the operation cannot be processed at this instant. A socket becomes nonblocking for all subsequent network calls by issuing a SIOCTL call with the FIONBIO option set.

One exception to this is the MSG_USRBUF option. For more details on how this option works on nonblocking sockets, see the individual descriptions for the SEND and RECV calls.

The DNP returns status information to your application in the *io_status* and *io_errno* fields of the NIOCB data structure immediately after processing your call.

6.4.1.3 Asynchronous Mode

If your application issues a call in asynchronous mode, your application continues executing while DNP processes the call. Unlike a nonblocking socket, asynchronous calls will always be accepted and queued for later processing.

If you issue commands in the asynchronous mode, you must code your application to check for completion by either:

- Specifying a callback routine which will be executed when the command completes
- or
- Polling the *io_status* field NIOCB for completion.

Upon entry to your callback routine, the following environment will be true:

- The routine will be running as part of DNP's background processing. The execution may interrupt the user's program as a response to network interrupt I/O completion.
- The stack will be valid and it will be the DNP's stack. You may change to a private stack if you wish.
- The address of the NIOCB will be on the stack and in registers DS:DX and ES:BX.

- The registers AX, BX, CX, SI, and DI do not have to be saved; they can be used by the callback routine.
- The callback routine should return, using the original stack, with a far (inter-segment) return.

Use the following guidelines when writing callback routines to prevent your system from crashing.

- The callback routine can interrupt your mainline program. The routine must be careful not to modify static data without interlocking with the mainline.
- The callback routine should not issue any calls to MS-DOS, since this may interrupt a call already in progress.
- The callback routine should not issue synchronous NIOCB calls, since DNP is currently processing this function's completion and will not be able to service the new one. Asynchronous and MSG_USRWAIT flagged calls are acceptable and the function will be queued for later processing.
- The callback routine should execute for a minimum amount of time because, while it is processing, the callback is preventing the DNP from further processing.

6.4.2 Setting the *io_flags* Field

You use the 2-byte *io_flags* field to specify function options. You can specify more than one option in this field. Bits for asynchronous I/O and callback routines can be used in conjunction with other bits for "peeking" at messages and for sending and receiving a multi-part message. Information on how you use bits with a specific function call appears in the individual call descriptions. You can specify the following flag options in the *io_flags* field. These are defined in the <socket.h> header file.

0x0001	MSG_OOB	Process out-of-band messages with the SEND and RECV calls.
0x0002	MSG_PEEK	Read the next pending message without removing the message from the receive queue.

0x0008	MSG_ASYNC	Queue this request and complete it asynchronously.
0x0010	MSG_CALLBACK	Specifies that the network will issue a callback routine when the function call completes.
0x0020	MSG_NEOM	For sending or receiving a single message in multiple operations. For sending, this flag indicates that this data is not the end of the message. For receiving, this flag indicates that the remaining message data should be saved for a later call.
0x0040	MSG_NBOM	For sending a single message in multiple operations. This flag indicates that this data is not the beginning of the message. If MSG_NEOM is also set, MSG_NBOM indicates the middle portion of the message.
0x0080	MSG_NIOCB	Flag to indicate new NIOCB must be set.
0x0100	MSG_USRBUF	Flag to indicate use of user-supplied buffer for sending or receiving large data buffers (up to 64KB).
0x0200	MSG_USRWAIT	This flag tells DNP not to wait for completion of an operation on a blocking socket. The user must check the <i>io_status</i> field for completion status.

6.4.3 Using Socket Numbers with Network Process Interface Calls

The network process uses socket numbers to identify a particular I/O session. You must assign a socket number before you can proceed with further network I/O. To assign a socket number, issue the ATTACH or ACCEPT function call, supplying the socket number in the *io_socket* field of the NIOCB. The assigned socket number will be used in the NIOCB to identify the socket on successive calls. If you specify a value of zero for this field, the DNP assigns a socket number for you.

6.5 Network Process Interface Calls

The DNP interface function calls allow you to send data to the DNP. Your assembly language program passes an NIOCB to the network process with each call. Use these calls to initiate or accept a connection, send or receive data, and create or disconnect a socket. Table 6-2 summarizes the DNP calls described in this chapter.

Table 6–2: DECnet-DOS Network Process Calls

Function Name	Function Code	Description
ABORT	10	Disconnect all logical links (if any) and detach the sockets that do not have the option SO_KEEPAIVE set.
ACCEPT	5	Accept an incoming connection request on a socket, and return a socket number.
ATTACH	0	Create a socket and attach a socket number.
BIND	2	Assign an object name or number to a socket.
CANCEL	20	Cancel previous function request.
CONNECT	4	Initiate a connection request on a socket.
DETACH	1	Disconnect all associated active logical links, and detach the specified socket and any associated sockets, only if the option SO_KEEPAIVE is not set.
DISCONNECT	6	Disconnect from the peer socket, and terminate the logical link connection.
GETSOCKOPT	26	Get the information associated with the socket options.
LISTEN	3	Listen for pending connections on a socket.
LOCALINFO	22	Retrieve network information for the local node.
PEERADDR	16	Retrieve information about your peer socket.
RCVD	8	Receive data on a specified socket.
RCVOOB	13	Receive out-of-band messages on a specified socket.
SELECT	23	Check the I/O status of the network sockets.
SEND	9	Send data on a specified socket.
SENDOOB	14	Send out-of-band messages on specified socket.
SETSOCKOPT	25	Set options associated with sockets.
SHUTDOWN	7	Shut down part or all of a full duplex logical link connection.
SIOCTL	24	Control the operations of open sockets.
SOCKADDR	15	Retrieve information set by the <i>BIND</i> call for the specified socket.

The following sections describe the DNP interface calls in detail. Each call description provides a summary of NIOCB fields required for input and output; any additional data structures, if required; and a summary of status messages that the DNP may return for each call. For detailed descriptions of the data structures, symbols, and options discussed in these sections, refer to Appendixes A through C as follows:

- Appendix A – for definitions of socket types and option flags.
- Appendix B – for definitions of the data structures and related data members that you use with the network process calls.
- Appendix C – for a summary of the error completion codes and descriptions.

ABORT

ABORT all logical links and detach sockets that do not have the option `SO_KEEPALIVE` set.

DESCRIPTION

Use this command to abort all logical links and detach the sockets that do not have the option `SO_KEEPALIVE` set. The ABORT call will not take effect if the `SO_KEEPALIVE` option is set. The socket remains attached and can maintain any active links.

To abort a socket that has `SO_KEEPALIVE` set, issue a `SETSOCKOPT` call with the NOT `SO_KEEPALIVE` condition set, before aborting a logical link. Set the `SO_KEEPALIVE` condition to the NOT (or off) state by preceding `SO_KEEPALIVE` with a tilde (~), as in `~SO_KEEPALIVE`. The NOT condition is the default for this socket option.

See also the `DISCONNECT` and `DETACH` call descriptions.

Input Data

Header Fields:

`io_fcode` = 10 (ABORT function code).

Parameter Fields:

`io_socket` must be zero.

`io_flags` set as follows:

<code>MSG_ASYNC</code> (0x0008)	for asynchronous mode, optional.
<code>MSG_CALLBACK</code> (0x0010)	to specify a callback, optional.
<code>MSG_NIOCB</code> (0x0080)	must be set.
<code>MSG_USRWAIT</code> (0x0200)	to specify a user wait loop, optional.

ABORT

io_psize must be zero.

Output Data

Header Fields

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno error detail. Errors returned for ABORT call:

[E2BIG] *io_psize* is too large.

[EINVAL] invalid function code for domain.

[ENETDOWN] network executor state is OFF.

[ENOBUFFS] no buffers available to handle request.

[USERABORT] link aborted by user.

ACCEPT

ACCEPT — accept an incoming connection request on a socket and return a socket number.

DESCRIPTION

Use this call to process the first connection request on the listening queue. DNP creates a new socket with the same properties as the listening socket. DNP returns the new socket number to the *io_socket* field. The listening socket remains open.

If the socket is set to nonblocking I/O and there are no queued connection requests, the DNP will return the value -1, "error completing call" status message to the *io_status* field. The *io_errno* field will contain the EWOULDBLOCK code.

Your application can accept incoming connection requests in one of two modes: immediate or deferred. Immediate mode is the default state; your application accepts a connection request immediately. You can change the accept mode to the deferred state by setting the SETSOCKOPT call. The deferred mode allows your application to examine access control information or optional user data before accepting or rejecting the request.

See the SETSOCKOPT call description and the descriptions for the DSO_ACCEPTMODE, DSR_CONNECT, and DSO_CONREJECT options for more information.

Input Data

Header Fields:

<i>io_fcode</i> = 5	(ACCEPT function code).
<i>io_socket</i>	nonzero number specifies the new socket number on which to receive the new session. Supply a zero to have DNP assign the socket number.

ACCEPT

io_flags set as follows:

MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize buffer size of the two additional data structures required for this call: *attach_dn* and *sockaddr_dn*. Specify the larger of the two, *sockaddr_dn* at 26 bytes.

io_callback the 4-byte address that is a far pointer to the callback routine. Use if MSG_CALLBACK is set in the *io_flags* field.

Parameter Fields:

attach_dn attach function data structure. Structure contains the following data fields:

<i>att_socket</i>	socket numbers of the listening socket.
<i>att_domain</i>	specifies communication domain as AF_DECnet.
<i>att_type</i>	specifies the socket type.
<i>att_protocol</i>	specifies the DECnet option.
<i>att_srp</i>	not used. Must be zero.
<i>att_supreq</i>	not used. Must be zero.

Output Data

Header Fields:

io_socket socket number. If the input *io_socket* was zero, this number will be filled in with the assigned socket number.

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the ACCEPT call:

[E2BIG] *io_psize* is too large.

[EBADF] not a valid socket number in *io_socket*.

[ECONNABORTED] the peer task has disconnected and the connection was aborted.

[EEXIST] socket number in *att_socket* already in use.

[EINVAL] not a valid socket number.

[EMFILE] no more available sockets.

[ENETDOWN] the network executor is state OFF.

[ETIMEDOUT] connection establishment was timed out before a connection was established.

[EUSERABORT] link aborted by user.

[EWOULDBLOCK] (synchronous only) no incoming connection request queued at this time.

ACCEPT

Parameter Fields:

<i>sockaddr_dn</i>	pointer to the socket address data structure. DNP returns data to the following fields:
<i>sdn_family</i>	specifies address family as AF_DECnet.
<i>sdn_objnum</i>	object number for the connecting task. Will be a number from 0 to 255.
<i>sdn_objname1</i>	specifies the size of the object name for the connecting task.
<i>sdn_objname</i>	16-byte array specifies the name of the object task. Used only when <i>sdn_objnum</i> equals zero.
<i>sdn_add</i>	name of node address data structure, format is struct <i>dn_naddr</i> .

ATTACH

ATTACH — create a socket and assign a socket number.

DESCRIPTION

The ATTACH call creates a socket and assigns a socket number.

Input Data

Header Fields:

- io_fcode* = 0 (ATTACH function code).
- io_socket* socket number. If you specify a zero, DNP assigns a socket number for you and returns the number to this field. If you specify a nonzero value, the network process assigns that value. The assigned socket number is used in subsequent network calls.
- io_flags* set as follows:
- | | |
|-----------------------|--|
| MSG_ASYNC (0x0008) | for asynchronous mode, optional. |
| MSG_CALLBACK (0x0010) | to specify a callback, optional. |
| MSG_NIOCB (0x0080) | must be set. |
| MSG_USRWAIT (0x0200) | to specify a user wait loop, optional. |
- io_psize* indicates buffer size of required data structure *attach_dn*. Must be 12 bytes.

Parameter Fields:

- attach_dn* pointer to address of the attach function data structure. Structure contains the following data fields:
- | | |
|-------------------|--|
| <i>att_socket</i> | ignored. Must be zero. |
| <i>att_domain</i> | specifies address family as AF_DECnet. |

ATTACH

<i>att_type</i>	specifies the socket type: SOCK_STREAM or SOCK_SEQPACKET.
<i>att_protocol</i>	specifies DECnet option for the socket, DNPROTO_NSP.
<i>att_srp</i>	not used. Must be zero.
<i>att_supreq</i>	not used. Must be zero.

Output Data

Header Fields:

io_socket socket number. If the input *io_socket* was zero, this number will be filled in with the assigned socket number.

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the ATTACH call:

[E2BIG]	<i>io_psize</i> is too large.
[EAFNOSUPPORT]	invalid <i>att_domain</i> .
[EEXIST]	socket number already in use.
[EINVAL]	not a valid socket number.
[EINVAL]	invalid function code for domain.
[EMFILE]	no more available sockets.

- [ENETDOWN] the network executor is state OFF.
- [ENOBUFS] no buffers available to handle request. Socket cannot be created. No more logical links.
- [ESOCKNOSUPPORT] *att_type* not STREAM or SEQSOCKET.
- [EUSERABORT] link aborted by user.

BIND

BIND

BIND — assign an object name or number to a socket.

DESCRIPTION

The **BIND** call assigns an object name or number to a socket. When a socket is first created with the **ATTACH** call, it exists in a name space but has no assigned name. The **BIND** call is used primarily by server tasks. The object name is required before a server task can listen for incoming connection requests using the **LISTEN** call. It can also be used by client tasks to identify themselves to server tasks. See also the **ACCEPT**, **CONNECT**, **PEERADDR**, and **SOCKADDR** call descriptions.

NOTE

The VMS proxy access by user name is made possible if the client task uses the **BIND** call specifying a user name as the object name. You should refer to the **SO_REUSEADDR** option of the **SETSOCKOPT** call if you wish to make more than one proxy connection with the same name.

Input Data

Header Fields:

<i>io_fcode</i> = 2	(BIND function code).
<i>io_socket</i>	socket number created on the ATTACH call.
<i>io_flags</i>	set as follows:
MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.
MSG_NIOCB (0x0080)	must be set.

io_psize 26 bytes, the size of the data structure *sockaddr_dn*.

Parameter Fields

<i>sockaddr_dn</i>	specifies the socket address data structure. A user fills in the data for each field. The same data members must be used with the ACCEPT call.
<i>sdn_family</i>	specifies the address family as AF_DECnet.
<i>sdn_flags</i>	must be zero.
<i>sdn_objnum</i>	object number for target network task. Specify a number from 0 to 255.
<i>sdn_objnamel</i>	size of the object name.
<i>sdn_objname</i>	the object name of the target network task. Use only when <i>sdn_objnum</i> equals zero.
<i>sdn_add</i>	ignored. Must be zero.

Output Data

Header Fields

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for BIND call:

[E2BIG] *io_psize* is too large.

[EADDRINUSE] specified name already used by another socket.

BIND

[EBADF]	not a valid socket number in <i>io_socket</i> .
[EINVAL]	invalid length for object name.
[ENETDOWN]	the network executor is state OFF.
[EUSERABORT]	link aborted by user.

CANCEL

CANCEL — cancel a previous asynchronous function call request.

DESCRIPTION

The **CANCEL** call allows a network process to cancel a previous asynchronous I/O request. This function call is used with the asynchronous form of the **ACCEPT**, **RCVD**, **RCVOOB**, **SELECT**, and **SEND** function calls. To cancel a previous function request, you must specify the socket number for that call. If you want to cancel a **SELECT** call, you must set the socket number to 0. Otherwise, the call will not be canceled.

There are two return values for any cancel operation:

- When a function call is canceled, the **CANCEL** function always returns success. It does not matter whether the previous request existed, was already completed, or was still in progress.
- To determine if the previous request was found and successfully canceled, you must examine *io_errno* for that particular call. An error code of **EINTR** will indicate successful cancellation. (See the **DIAGNOSTICS** section for each applicable function call.)

Input Data

Header Fields:

io_code = 20 (CANCEL function code).

io_socket specifies the socket number that must match the socket number on the call you want to cancel. Specify zero to cancel a **SELECT** call.

io_flags set as follows:

MSG_ASYNC (0x0008) for asynchronous mode, optional.

MSG_CALLBACK (0x0010) to specify a callback, optional.

BIND

MSG_NIOCB (0x0080) must be set.

MSG_USRWAIT (0x0200) to specify a user wait loop, optional.

io_psize specifies size of *io_buffer* as 4 bytes.

Parameter Fields:

io_buffer address of the NIOCB for the call you are canceling.

Output Data

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the CANCEL call:

[E2BIG] *io_psize* is too large.

[EBADF] invalid socket

[EINVAL] invalid function code for domain.

[ENOBUFFS] no buffer space available. Socket cannot be created. No more logical links.

[ENETDOWN] the network executor is state OFF.

[EUSERABORT] link aborted by user.

CONNECT

CONNECT — initiate a connection request on a socket.

DESCRIPTION

Use the CONNECT call to issue a connection request to another socket. You can pass optional data, such as access control information, to the peer task on this call. This data must be previously set by the SETSOCKOPT call. To issue subsequent CONNECT calls, your application must reissue the SETSOCKOPT call to set up new optional data and/or access control information.

NOTE

Subsequent connection requests cannot be made on the same socket until it has been disconnected.

If nonblocking I/O mode is set and the CONNECT call is issued, DNP will return the value -1, "error completing call" status message to the *io_status* field. The *io_errno* field will contain the EINPROGRESS code.

Input Data

Header Fields:

<i>io_fcode</i> = 4	(CONNECT function code).
<i>io_socket</i>	socket number created by the ATTACH call. This socket number is used for establishing a connection between the user tasks. It is also used with subsequent send and receive function calls.
<i>io_flags</i>	set as follows:
MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.

CONNECT

MSG_USRWAIT (0x0200) to specify a user wait loop, optional.

io_psize specifies the size of the data structure *sockaddr_dn* required for this call as 26 bytes.

io_callback the 4-byte address that is a far pointer to the callback routine. Use if **MSG_CALLBACK** is set in *io_flags* field.

Parameter Fields:

sockaddr_dn socket address data structure. The following fields must be filled in:

sdn_family specifies the address family as **AF_DECnet**.

sdn_flags must be zero.

sdn_objnum object number for target network task. Specify a number from 0 to 255.

sdn_objname1 size of the object name.

sdn_objname the object name of target network task. Use only when *sdn_objnum* equals zero.

sdn_add specifies the node address structure *dn_naddr* for the server task.

Output Data

Header Fields

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

CONNECT

io_errno contains additional error detail. Errors returned for CONNECT call:

[E2BIG]	<i>io_psize</i> is too large.
[EAFNOSUPPORT]	addresses in the specified address family cannot be used with this particular socket.
[EBADF]	the argument <i>io_socket</i> does not contain a valid socket number.
[EBUSY]	the socket is not in idle state. The socket is in the process of being connected or disconnected; the socket is a connected or listening socket.
[ECONNABORTED]	the peer task has disconnected and the connection was aborted.
[ECONNREFUSED]	the attempt to connect was explicitly rejected by the target system or task.
[ECONNRESET]	the remote task has disconnected the link.
[EHOSTUNREACH]	the remote node is unreachable.
[EINPROGRESS]	the connection request is now in progress.
[EINVAL]	the object name of the server task is too long.
[ENETDOWN]	the local network is OFF.
[ENETUNREACH]	the network cannot be reached from this host.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.
[ERANGE]	the object number of the server task is invalid. The valid range is from 0 to 255.

CONNECT

[ESRCH] the server object does not exist on the remote node.

[ETIMEDOUT] connection establishment was timed out before a connection was established. The remote node may not be on the network.

[ETOOMANYREFS] the remote node has accepted the maximum number of connection requests.

DETACH

DETACH—disconnect the associated active session and detach the specified socket and any associated sockets, only if the socket option **SO_KEEPALIVE** is not set.

DESCRIPTION

Use the **DETACH** call to disconnect the socket and detach all active sessions associated with this socket. Identify the socket you wish to **DETACH** by supplying the socket number that was returned on the **ACCEPT** or **ATTACH** call.

If the **SO_LINGER** socket option is set (the default condition), a **DISCONNECT** operation will not close the active link until all queued transmit data is acknowledged. If you clear the **SO_LINGER** socket option by setting the **SO_DONTLINGER** option, a **DISCONNECT** call will immediately take down the link with a lower level abort reason.

The **DETACH** call will not take effect if the **SO_KEEPALIVE** option is set for this socket. The socket remains attached and maintains any active links associated with it.

If you previously set the socket option to **SO_KEEPALIVE**, you must issue a **SET-SOCKOPT** call with the **NOT SO_KEEPALIVE** condition set, before issuing the **DETACH** call. Set the **SO_KEEPALIVE** condition to the **NOT** (or off) state by preceding **SO_KEEPALIVE** with a tilde (~), as in **~SO_KEEPALIVE**. The **NOT** condition is the default for this socket option.

Input Data

Header Fields:

io_fcode = 1 (DETACH function code).

Parameter Fields:

io_socket socket number created on the **ACCEPT** or **ATTACH** call.

DETACH

io_flags set as follows:

MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize must be zero, no parameter list used for this call.

Output Data

Header Fields:

io_status completion status field:

- 0 = call completed successfully
- 1 = error completing call
- 2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for DETACH call:

[E2BIG]	<i>io_psize</i> is too large
[EACCES]	socket has SO_KEEPALIVE set.
[EBADF]	not a valid socket number.
[ENOBUFFS]	no buffer space available. Socket cannot be created. No more logical links.

DISCONNECT

DISCONNECT the session from the peer.

DESCRIPTION

Use the DISCONNECT call to disconnect the session on the socket from the peer.

The DISCONNECT call will not take effect if the SO_KEEPALIVE option is set for this socket. The socket remains attached and maintains any active links associated with it.

If you previously set the socket option to SO_KEEPALIVE, you must issue a SETSOCKOPT call with the NOT SO_KEEPALIVE condition set, before issuing the DISCONNECT call. Set the SO_KEEPALIVE condition to the NOT (or off) state by preceding SO_KEEPALIVE with a tilde (~), as in ~SO_KEEPALIVE. This is the default for this socket option.

The effect DISCONNECT has on unsent data queued for a remote task depends on the linger option set on the SETSOCKOPT function call. If SO_LINGER is set, control is returned to the task, but the link is not disconnected until the queued data is sent (graceful disconnect). If SO_DONTLINGER is set, control is returned to the task, and any queued data is lost.

Input Data

Header Fields

io_fcode = 6 (DISCONNECT function code).

Parameter Fields:

io_socket socket number created on the ACCEPT or ATTACH call.

io_flags set as follows:

MSG_ASYNC (0x0008) for asynchronous mode, optional.

DISCONNECT

MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize must be zero.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for DISCONNECT call:

[E2BIG]	<i>io_psize</i> is too large.
[EACCES]	socket has SO_KEEPALIVE option set.
[EBADF]	not a valid socket number.
[ENOBUFFS]	no buffer space available. Socket cannot be created. No more logical links.

GETSOCKOPT

GETSOCKOPT — get the options associated with sockets.

DESCRIPTION

Use the GETSOCKOPT call to manipulate various options associated with a socket. Options exist at multiple levels; therefore you must specify the level number for the desired operation. In the following discussion, references are made to symbolic values. Refer to Appendix A for a full description of the socket options.

At the socket level (SOL_SOCKET), the options include:

- **SO_KEEPAIVE**. If this option is set on a socket, any links and sockets associated with this socket will remain active, despite any attempts to disconnect them.

Before terminating a socket connection, you must issue another SETSOCKOPT call to change this socket option. Set SO_KEEPAIVE to the NOT condition by preceding it with a tilde (~). ~SO_KEEPAIVE is the default condition.

- **SO_LINGER** controls the actions taken when unsent messages are queued on a socket and a DISCONNECT call is issued. If SO_LINGER is set, the connection is maintained until the outstanding messages have been sent. This is the default condition.
- **SO_DONTLINGER** also controls the actions of unsent messages. If SO_DONTLINGER is set, and the DISCONNECT call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.
- **SO_REUSEADDR** allows the reuse of a name already bound to a socket. For most situations, a name is bound to a socket only once. However, this option enables you to reuse the same name. This particular option **must** be used for outgoing connection requests only. It cannot be used for incoming connections.

GETSOCKOPT

VMS proxy access by user name is made possible if the client task uses the **BIND** call specifying the user name as the object name. If you wish to make more than one proxy connection with the same user name, you must use the **SO_REUSEADDR** option.

- **SO_RCVUSRBUF** tells DNP that the user is using **MSG_USRBUF** receives and that select functions (with readfds selected) should complete upon receipt of the first segment of a message. Otherwise, select does not indicate that the message is read-ready until the end of the message is received.

At the DECnet level (**DNPROTO_NSP**), socket options may specify the way in which a connection request is accepted or rejected, may be used to set up optional user data and/or access control information, and may be used to obtain current link state information. The following socket options can be specified:

- **DSO_ACCEPTMODE**. The accept option mode is used at the DECnet level for processing **ACCEPT** calls. A socket must be bound before specifying this option. You can supply the following values for this option.
 - **ACC_IMMED** mode is the default condition for this option. When immediate mode is in effect, control is immediately returned to the server task following an **ACCEPT** call with the connection request accepted. The access control information and/or optional user data is ignored by the server task.
 - **ACC_DEFER** mode indicates that the server task completes the **ACCEPT** call without fully completing the connection to the client task. In this case, the server task can examine the optional access control or user data before it decides to accept or reject the connection request. The server task can then issue the **SETSOCKOPT** call with the appropriate reject or accept option.
 - **ACC_REUSE** enables your application to reuse the listening socket to receive incoming connections. To receive the connect, your application must issue an **ACCEPT** call. Your application must supply the socket number (**iocb.io_socket**) either as the same value as the listening socket or as a zero that will return the correct socket.

GETSOCKOPT

This mode can be used along with `ACC_IMMED` or `ACC_DEFER`. The argument to `DSO_ACCEPTMODE` should be the OR of the values (such as `ACC_DEFER | ACC_REUSE`). If the connection is rejected or disconnected, the socket must be explicitly reenabled for listening by issuing a new `LISTEN` call.

NOTE

This call is intended primarily for use by single-threaded servers. From the time a single connect is received on the socket until another listen is posted, further connects are rejected with the message "Object unknown." Another listen must be posted quickly, but that does not close the timing window. The client must be aware of the possible rejection and retry.

- **DSO_CONDATA** allows up to 16 bytes of optional user data to be set by the `SETSOCKOPT` call. It can be sent as a result of the `CONNECT` or the `ACCEPT` (with the deferred option) calls. The optional data is passed in a structure of type `optdata_dn`. The data is read by the task issuing the `GETSOCKOPT` call with this option.

- **DSO_DISDATA** allows up to 16 bytes of optional data to be set by the `SETSOCKOPT` call. It can be sent as a result of the `DISCONNECT` call. The optional data is passed in a structure of type `optdata_dn`.

The data is read by the task issuing the `GETSOCKOPT` call with this option.

- **DSO_CONACCESS** allows access control information to be passed by the user task. This information is set with the `SETSOCKOPT` call. The access data is sent to the server task. It is passed with the `CONNECT` call in a structure of type `accessdata_dn`. The access data is read by the task issuing the `GETSOCKOPT` call with this option.
- **DSO_LINKINFO** determines the state of the logical link connection. When the `GETSOCKOPT` call is issued with this option, the state of the logical link is returned in a logical link information data structure, `linkinfo_dn`.

GETSOCKOPT

Input Data

Header Fields:

io_fcode = 26 GETSOCKOPT function code.

io_socket the number for a socket created on the ACCEPT and/or deferred mode ACCEPT or ATTACH call.

io_flags set as follows:

MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize specifies the size of the data structure *sockopt_dn* as 8 bytes.

Parameter Fields:

sockopt_dn specifies the socket option data structure.

The structure contains the following data fields:

sop_level 2 bytes; specifies the level at which options are manipulated.

If the level is set to SOL_SOCKET, then *sop_optval* and *sop_optlen* are ignored. If the level is set to DPROTO_NSP, then the rest of the data structure can contain either access control or optional user data, or access mode information.

sop_optname 2 bytes; specifies options to be interpreted.

When the socket level is set to DPROTO_NSP, *sop_optname* can be set to one of 6 specific options (for example, DSO_CONDATA).

GETSOCKOPT

sop_optval 4 bytes; specifies access option values used with the *sop_optlen* SETSOCKOPT and the GETSOCKOPT calls.

The interpretation of each argument is function dependent as shown here:

sop_optval specifies the pointer to a buffer which will contain the returned value for the requested option(s).

sop_optlen is a value result parameter. It should initially contain the size of the buffer pointed to by *sop_optval*. On return, it will contain the actual size of the returned value.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SETSOCKOPT call:

[E2BIG] *io_psize* is too large.

[EACCES] unable to disconnect the socket.

[EBADF] the argument *io_socket* does not contain a valid descriptor.

[ECONNABORTED] the client task disconnected before the ACCEPT call completed.

[EDOM] the acceptance mode is not valid.

[ENOBUFFS] no buffer space available. Socket cannot be created. No more logical links.

GETSOCKOPT

[ENOPROTOOPT]	there was no access control information supplied with the connection request.
[EOPNOTSUPP]	the option is unknown.
[ENETDOWN]	the local network is OFF.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.

Parameter Fields:

sop_optlen is a value result parameter. On return, it contains the actual size of the returned value for the buffer pointed to by *sop_optval*.

LISTEN

LISTEN — listen for incoming connections on a socket.

DESCRIPTION

Use the **LISTEN** call to declare that the socket is available as a server for client connections. The server task must issue the **LISTEN** before it can accept or reject an incoming connection. The server task must issue a **BIND** call to assign a name or number to the socket before it can listen for incoming client connections.

See also the descriptions for the **ACCEPT** and **SELECT** calls.

If you detach a listening socket while the socket is receiving client connections, all links associated with that listening socket immediately abort and all outstanding data is lost.

Input Data

Header Fields:

<i>io_fcode</i> = 3	(LISTEN function code).
<i>io_socket</i>	specifies a number for a bound socket.
<i>io_flags</i>	set as follows:
MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.
<i>io_psize</i>	specifies the size of the required data structure <i>listen_dn</i> as 2 bytes.

LISTEN

Parameter Fields:

listen_dn the listen data structure. The structure contains the following data field:

lsn_backlog defines the total maximum number of unaccepted incoming connects which are allowed on this particular socket. The maximum allowable number of incoming connects is 5. If a connection request arrives when the queue is full, the client task will receive the error ECONNREFUSED.

Output Data

Header Fields

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for LISTEN call:

[E2BIG] *io_psize* is too large.

[EBADF] the argument *io_socket* does not contain a valid socket number.

[EOPNOTSUPP] the specified socket type does not support the listen operation.

[ENOBUFFS] no buffer space available. Socket cannot be created. No more logical links.

LOCALINFO

LOCALINFO — retrieve network information for the local node.

DESCRIPTION

Use the LOCALINFO call to retrieve network information for the local node. DNP returns the following data to your application.

- The software version number for the network process
- The local node name and address
- The maximum possible segment buffer size which can be used on a logical link
- The number of sockets available for data exchange
- The current DECnet database device and path

Input Data

Header Fields:

io_fcode = 22 (LOCALINFO function code).

io_socket must be zero. Not used.

io_flags set as follows:

MSG_ASYNC (0x0008) for asynchronous mode, optional.

MSG_CALLBACK (0x0010) to specify a callback, optional.

MSG_NIOCB (0x0080) must be set.

MSG_USRWAIT (0x0200) to specify a user wait loop, optional.

io_psize specifies the size of the required data structure *localinfo_dn* as 20 bytes.

LOCALINFO

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for LOCALINFO call:

[E2BIG] *io_psize* is too large.

[ENOBUFS] no buffer space available. Socket cannot be created. No more logical links.

Parameter Fields:

localinfo_dn local node information data structure. A user retrieves data from the fields DNP fills in on this function call.

The following data fields can be filled in by this function call:

lcl_version software version number for the network process.

lcl_nodename local node name.

lcl_nodeaddr local node address.

LOCALINFO

<i>lcl_segsize</i>	maximum buffer segment size to be used on the logical link.
<i>lcl_sockets</i>	number of sockets available for data exchange.
<i>lcl_decnet_device</i>	the DECnet database device.
<i>lcl_decnet_path</i>	the address of the buffer that contains the DECnet database path specification and device name.

PEERADDR

PEERADDR

PEERADDR — get information about your peer socket.

DESCRIPTION

Use the PEERADDR call to obtain information about the application on the other end of the connection.

Input Data

Header Fields:

<i>io_fcode</i> = 16	(PEERADDR function code).
<i>io_socket</i>	number for the socket created on the ACCEPT or ATTACH call.
<i>io_flags</i>	set as follows:
MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.
<i>io_psize</i>	specifies the size of the output data structure <i>sockaddr_dn</i> as 26 bytes.

Output Data

Header Fields:

io_status completion status field:
0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the PEERADDR call:

[E2BIG] *io_psize* is too large.

[EBADF] the argument *io_socket* does not contain a valid socket.

[ENETDOWN] the local network is OFF.

[ENOBUFS] no buffer space available. Socket cannot be created. No more logical links.

Parameter Fields:

sockaddr_dn specifies the socket address data structure. A user retrieves data from the fields DNP fills in on this function call.

The following data fields can be filled in by this function call:

sdn_family specifies the address family as AF_DECnet.

sdn_flags must be zero.

sdn_objnum object number for target network task. Can be a number from 0 to 255.

sdn_objname1 size of the object name.

sdn_objname the object name of target network task. Used only when *sdn_objnum* equals zero.

sdn_add specifies the node address structure *dn_naddr* for the server task.

RCVD

RCVD — receive data on a specified socket.

DESCRIPTION

Use the RCVD call to receive data from the peer task. If no messages are available at the socket, the RCVD call waits for a message to arrive unless the socket is nonblocking. In this case, a status of -1 is immediately returned with the field *io_errno* set to EWOULDBLOCK.

If the session becomes disconnected, queued data can still be received. However, if you shut down the socket or detach it, the queued data is lost. When the logical link is not in a connected state, and all data has been read, the RCVD call returns zero bytes.

For sequenced sockets, you can read a single message with multiple calls into multiple buffers. To do this, set the *io_flags* field to MSG_NEOM (not end of message).

NOTE

MSG_NEOM is ignored for receiving stream sockets.
Stream mode destroys all message boundaries.

For example, you want to receive a 300-byte message, but you only specified a receive buffer of 100 bytes. Normally, the RCVD call would discard the rest of the message (after you read the first 100 bytes) and return a status of 0. Setting *io_flags* to MSG_NEOM indicates that the caller wants to retain the unread data. If the user did not receive the entire message, *io_status* returns a value of 1. MSG_NEOM allows you to issue another RCVD call and read the remaining 200 bytes of the buffer, 100 bytes at a time.

In addition to flagging the RCVD call with MSG_NEOM, you can also set the flags option to MSG_PEEK. This option enables you to “peek” or read the next pending message without removing it from the receive queue.

A successful return with a length of zero indicates two different situations. Either you have received a message of zero length, or the link has been disconnected. To distinguish between the two, use the `GETSOCKOPT` function with the option `DSO_LINKINFO` set, to check if the link is still running. Once a link is disconnected, all subsequent `RCVD` and `SELECT` read-ready operations will return immediately.

The `MSG_USRBUF` flag gives you the ability to send and receive messages longer than the normal 4K byte restriction. This flag specifies that the user message buffer can be used to assemble or disassemble the message as it is being received or transmitted. During the operation of this call, the `NIOCB` and buffer cannot be used by the user program.

NOTE

`MSG_USRBUF` will cause a nonblocking socket to block during the operation. Use the `MSG_USRWAIT` option to allow concurrent program execution.

Use the `SELECT` call to determine when more data has arrived.

Input Data

Header Fields:

<i>io_fcode</i> = 8	(RCVD function code).
<i>io_socket</i>	the number for a socket created on the <code>ACCEPT</code> or <code>ATTACH</code> call.
<i>io_flags</i>	set as follows:
<code>MSG_ASYNC (0x0008)</code>	for asynchronous mode, optional.
<code>MSG_CALLBACK (0x0010)</code>	to specify a callback, optional.
<code>MSG_NEOM (0x0020)</code>	to retain unread data.
<code>MSG_NIOCB (0x0080)</code>	must be set.

RCVD

MSG_PEEK (0x0002)	to read next pending without removing it from the queue.
MSG_USRBUF (0x0100)	to receive large data messages (up to 64KB) to user buffers.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize the size of the structure. *buffer_dn*. must be set to 6.

io_callback the 4-byte address that is a far pointer to the callback routine. Use if MSG_CALLBACK is set in *io_flags* field.

Parameter Fields:

buffer_dn describes the user-supplied receive buffer. This data structure contains the following fields:

- io_buffer* contains the pointer to the user receive buffer.
- io_buflen* specifies the length of the user receive buffer.

Output Data

Header Fields:

io_status completion status field:

- 0 = call completed successfully
- 1 = error completing call
- 2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the RCVD call:

- [E2BIG] *io_psize* is too large.
- [EBADF] the argument *io_socket* does not contain a valid socket number.

[ENETDOWN]	the local network is OFF.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.
[EWOULDBLOCK]	the receive operation would block because there is currently no data to receive. (Asynchronous mode only.)

Parameter Fields:

buffer_dn specifies the receive buffer. This data structure contains the following fields:

io_buffer not used.

io_buflen specifies the length of the received data upon completion.

RCVOOB

RCVOOB

RCVOOB — receive out-of-band messages on a specified socket.

DESCRIPTION

Use the RCVOOB call to receive out-of-band data from the peer task. Out-of-band messages are available to the task independently of normal messages.

If there is no data waiting for your application to receive, DNP returns an “error completing call” status message and sets *io_errno* to EWOULDBLOCK. This occurs for both blocking and nonblocking calls.

If the session becomes disconnected, you can still receive queued data. However, if you shut down the socket or detach it, you cannot receive the queued data. When the logical link is not in a connected state, and all data has been read, the RCVOOB call will not return.

Your application can issue a SELECT call to determine when more data arrives.

Input Data

Header Fields:

<i>io_fcode</i> = 13	(RCVOOB function code).
<i>io_socket</i>	specifies the number for a socket created on the ACCEPT or CONNECT call.
<i>io_flags</i>	set as follows:
MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_PEEK (0x0002)	to read next pending without removing it from the queue.

MSG_USRWAIT (0x0200) to specify a user wait loop, optional.

io_psize specifies the size of the data structure **buffer_dn** as 6 bytes.

io_callback the 4-byte address that is a far pointer to the callback routine. Use if **MSG_CALLBACK** is set in **io_flags** field.

Parameter Fields:

buffer_dn specifies the receive buffer. This data structure contains the following fields:

io_buffer contains the pointer to the user receive buffer.

io_buflen specifies the length of the user receive buffer.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = no change, call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the RCVOOB call:

[EBADF] the argument **io_socket** does not contain a valid socket.

[EPIPE] the link has been disconnected, aborted, or shut down. No further messages can be sent.

[ENETUNREACH] the network cannot be reached from this host.

[EWOULDBLOCK] the receive operation would block because there is currently no data to receive.

RCVOOB

Parameter Fields:

buffer_dn specifies the receive buffer. This data structure contains the following fields:

io_buffer not used.

io_buflen specifies the length of the received data, in bytes, upon completion.

SELECT

SELECT — check the I/O status of the network sockets.

DESCRIPTION

Use the SELECT call to determine if the network sockets are ready for reading or writing, or if they have any outstanding out-of-band messages. Specify the socket you wish to examine with the *select_dn* data structure.

The SELECT call does not tell you if the logical link has been broken. You should use the SELECT call to help manage your ACCEPT, SEND, SENDOOB, RCVD, and RCVOOB calls.

The I/O descriptors are long words which contain bit masks. Each bit in a mask represents one socket number. For example, socket “3” is the fourth bit or has a hex value of 8.

NOTE

The SELECT call can check only socket numbers in the range 0 to 31.

To specify the bit for any socket number, use the value created by the ATTACH or the ACCEPT call, as “*I* < *s*”.

The SELECT call examines the network sockets until the call time out (see *sel_seconds*) or status is returned. If you specify multiple sockets to be examined, and one socket becomes detached, the SELECT call will return with the error message, EBADF. You must reissue the SELECT call to examine the remaining sockets.

Input Data

Header Fields:

<i>io_fcode</i> = 23	(SELECT function code).
<i>io_socket</i>	socket number. Must be zero.

SELECT

io_flags set as follows:

MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize specifies the size of the data structure *select_dn* as 16 bytes.

io_callback the 4-byte address that is a far pointer to the callback routine. Use if MSG_CALLBACK is set in *io_flags* field.

Parameter Fields:

select_dn is the select data structure that specifies the socket you want to examine. The user fills in data for each of the following fields:

<i>sel_nfds</i>	specifies the highest socket number to be checked. The bits from (1<<0) to (1<<(nfds-1)) are examined.
<i>sel_read</i>	specifies the socket numbers (as bit masks) to be examined for read ready. For listening sockets, a read ready condition indicates that an incoming connection request is available. For active sessions on sequenced sockets, there is a complete message to be read. For stream sockets, there is some data to be read. If a socket disconnects or aborts, a read ready condition will always occur so that the link state can be tested.

This descriptor can be given as a zero value if of no interest.

SELECT

sel_write specifies the socket numbers (as bit masks) to be examined for write ready. A write ready condition exists when the logical link is connected. This descriptor can be given as a zero value if of no interest.

sel_except specifies the socket numbers (as bit masks) to be examined for out-of-band data ready. There is a pending out-of-band data message to receive. This descriptor can be given as a zero value if of no interest.

sel_seconds defines the maximum interval to wait for a descriptor selection to be completed. If the time value is set to -1, the SELECT call will wait until an event occurs. If the time value equals 0, then the SELECT call will return after an immediate poll. If the time value is greater than zero, the SELECT call will return either after *n* seconds have expired, or when an event occurs, whichever one comes first.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SELECT call:

[EBADF] one of the specified bit masks is an invalid descriptor.

[ENETDOWN] the local network is OFF.

SELECT

[ENOBUFFS] no buffer space available. Socket cannot be created. No more logical links.

[E2BIG] *io_psize* is too large.

Parameter Fields:

sel_read this bit is set if socket is read ready, or cleared if not. If ready, DNP returns the socket numbers (as bit masks) to be examined.

sel_write this bit is set if socket is write ready, or cleared if not. If ready, DNP returns the socket numbers (as bit masks) to be examined.

sel_except this bit is set if socket is ready to RECEIVE out-of-band data, or cleared if not. If ready, DNP returns the socket numbers (as bit masks) to be examined.

SEND

SEND data on a specified socket.

DESCRIPTION

Use the SEND call to transmit data to your peer. If you cannot get enough buffer space on a blocking socket, the call is blocked. You must wait until current transmissions are finished. If the socket is set to nonblocking, DNP returns the “error completing call” status message to *io_status* and the error value EWOULDBLOCK to *io_errno*. If a socket disconnects, any outstanding data to be sent is discarded.

For sequenced sockets, you can send a single message in multiple calls. To do this, you should flag the SEND call with the required message options, NEOM (not end of message) and NBOM (not beginning of message).

The following example describes how to send a three-part message and have the SEND call treat it as one message.

The *io_flags* are set as follows:

- First buffer (*io_flags* = MSG_NEOM)
- Second buffer (*io_flags* = MSG_NEOM and MSG_NBOM)
- Third buffer (*io_flags* = MSG_NBOM)

At the receiving end, the three-part message will be reconstructed and treated as a single message.

Input Data

Header Fields

io_fcode = 9 (SEND function code).

io_socket specifies the number for a socket created on the ACCEPT or the ATTACH call.

SEND

io_flags set as follows:

MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NBOM (0x0040)	to indicate not beginning of message.
MSG_NEOM (0x0020)	to send a single message with multiple calls.
MSG_NIOCB (0x0080)	must be set.
MSG_USRBUF (0x0100)	to send large data buffers (up to 64KB).
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize specifies the size of the data structure *buffer_dn* as 6 bytes.

io_callback the 4-byte address that is a far pointer to the callback routine. Use if MSG_CALLBACK is set in *io_flags* field.

Parameter Fields:

buffer_dn describes the send buffer. This data structure contains the following fields:

io_buffer contains the pointer to the user send buffer.

io_buflen specifies the length of the user send buffer.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

SEND

-2 = no change, call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SEND call:

[EBADF]	one of the specified bit masks is an invalid descriptor.
[EMSGSIZE]	the size of the outgoing message is more than 4096 bytes.
[ENOTCONN]	the SEND call did not complete and the link was disconnected.
[EPIPE]	the link has been disconnected, aborted, or shut down. No further messages can be sent.
[EWOULDBLOCK]	(NONBLOCKING I/O ONLY) the socket is marked for nonblocking and no connections are waiting to be accepted.

buffer_dn specifies the receive buffer. This data structure contains the following fields:

io_buffer not used.

io_buflen specifies the length of the transferred data upon completion.

SENDOOB

SENDOOB

SENDOOB — send out-of-band messages on a specified socket.

DESCRIPTION

Use the SENDOOB call to send out-of-band data to your peer. An out-of-band message is a high priority message that you can send to the peer task. DNP processes out-of-band messages before it processes normal data. If a socket disconnects, any outstanding data to be sent is lost.

Input Data

Header Fields:

- io_fcode* = 14 (SENDOOB function code).
- io_socket* the number for a socket created on the ACCEPT or ATTACH call.
- io_flags* set as follows:
- | | |
|-----------------------|--|
| MSG_ASYNC (0x0008) | for asynchronous mode, optional. |
| MSG_CALLBACK (0x0010) | to specify a callback, optional. |
| MSG_NIOCB (0x0080) | must be set. |
| MSG_USRWAIT (0x0200) | to specify a user wait loop, optional. |
- io_psize* size of the *buffer_dn*.data structure as as 6 bytes.
- io_callback* the 4-byte address that is a far pointer to the callback routine. Use if MSG_CALLBACK is set in *io_flags* field.

Parameter Fields:

- buffer_dn* describes the receive buffer. This data structure contains the following fields:

io_buffer contains the pointer to the user receive buffer.

io_buflen specifies the length of the user receive buffer.

Output Data

Header Fields:

io_buflen specifies the number of transferred bytes upon successful completion.

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SENDOOB call:

[EALREADY]	the out-of-band message could not be sent. A previous transmission request is still in progress.
[EBADF]	the argument <i>io_socket</i> does not contain a valid socket number.
[EMSGSIZE]	the size of the outgoing message is more than 16 bytes.
[ENOTCONN]	the SEND call did not complete and the link was disconnected.
[EPIPE]	the link has been disconnected, aborted, or shut down. No further messages can be sent.
[EWOULDBLOCK]	(NONBLOCKING I/O ONLY) message cannot be sent yet because previous message has not been read.

SETSOCKOPT

SETSOCKOPT

SETSOCKOPT — set the options associated with sockets.

DESCRIPTION

Use the SETSOCKOPT call to manipulate various options associated with a socket. Options exist at multiple levels; therefore you must specify the level number for the desired operation. In the following discussion, references are made to symbolic values. Refer to Appendix A for a full description of the socket options.

At the socket level (SOL_SOCKET), the options include:

- **SO_KEEPALIVE**. If this option is set on a socket, any links and sockets associated with this socket will remain active, despite any attempts to disconnect them.

Before terminating a socket connection, you must issue another SETSOCKOPT call to change this socket option. Set SO_KEEPALIVE to the NOT condition by preceding it with a tilde (~). ~SO_KEEPALIVE is the default condition.

- **SO_LINGER** controls the actions taken when unsent messages are queued on a socket and a DISCONNECT call is issued. If SO_LINGER is set, the connection is maintained until the outstanding messages have been sent. This is the default condition.
- **SO_DONTLINGER** also controls the actions of unsent messages. If SO_DONTLINGER is set, and the DISCONNECT call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.
- **SO_REUSEADDR** allows the reuse of a name already bound to a socket. For most situations, a name is bound to a socket only once. However, this option enables you to reuse the same name. This particular option **must** be used for outgoing connection requests only. It cannot be used for incoming connections.

SETSOCKOPT

VMS proxy access by user name is made possible if the client task uses the BIND call specifying the user name as the object name. If you wish to make more than one proxy connection with the same user name, you must use the SO_REUSEADDR option.

- **SO_RCVUSRBUF** tells DNP that the user is using MSG_USRBUF receives and that select functions (with readfds selected) should complete upon receipt of the first segment of a message. Otherwise, select does not indicate that the message is read-ready until the end of the message is received.

At the DECnet level (DNPROTO_NSP), socket options may specify the way in which a connection request is accepted or rejected, may be used to set up optional user data and/or access control information, and may be used to obtain current link state information. The following socket options can be specified:

- **DSO_ACCEPTMODE**. The accept option mode is used at the DECnet level for processing ACCEPT calls. A socket must be bound before specifying this option. You can supply the following values for this option.
 - **ACC_IMMED** mode is the default condition for this option. When immediate mode is in effect, control is immediately returned to the server task following an ACCEPT call with the connection request accepted. The access control information and/or optional user data is ignored by the server task.
 - **ACC_DEFER** mode indicates that the server task completes the ACCEPT call without fully completing the connection to the client task. In this case, the server task can examine the optional access control or user data before it decides to accept or reject the connection request. The server task can then issue the SETSOCKOPT call with the appropriate reject or accept option.
 - **ACC_REUSE** enables your application to reuse the listening socket to receive incoming connections. To receive the connect, your application must issue an ACCEPT call. Your application must supply the socket number (iocb.io_socket) either as the same value as the listening socket or as a zero that will return the correct socket.

SETSOCKOPT

This mode can be used along with `ACC_IMMED` or `ACC_DEFER`. The argument to `DSO_ACCEPTMODE` should be the OR of the values (such as `ACC_DEFER | ACC_REUSE`). If the connection is rejected or disconnected, the socket must be explicitly reenabled for listening by issuing a new `LISTEN` call.

NOTE

This call is intended primarily for use by single-threaded servers. From the time a single connect is received on the socket until another listen is posted, further connects are rejected with the message "Object unknown." Another listen must be posted quickly, but that does not close the timing window. The client must be aware of the possible rejection and retry.

- **DSO_CONACCEPT** allows the server task to accept the pending connection on the socket returned by the `ACCEPT` call. The original listening socket was set to deferred accept mode. Any optional user data previously set by `DSO_CONDATA` will also be sent.
- **DSO_CONACCESS** allows access control information to be passed by the user task. This information is set with the `SETSOCKOPT` call. The access data is sent to the server task. It is passed with the `CONNECT` call in a structure of type *accessdata_dn*. The access data is read by the task issuing the `GETSOCKOPT` call with this option.
- **DSO_CONDATA** allows up to 16 bytes of optional user data to be set by the `SETSOCKOPT` call. It can be sent as a result of the `CONNECT` or the `ACCEPT` (with the deferred option) calls. The optional data is passed in a structure of type *optdata_dn*. The data is read by the task issuing the `GETSOCKOPT` call with this option.
- **DSO_CONREJECT** allows the server task to reject the pending connection on the socket returned by the `ACCEPT` call. The original listening socket was set to deferred accept mode. Any optional user data previously set by `DSO_DISDATA` will also be sent. The reject reason is the value passed with this option.

SETSOCKOPT

- **DSO_DISDATA** allows up to 16 bytes of optional data to be set by the SETSOCKOPT call. It can be sent as a result of the DISCONNECT call. The optional data is passed in a structure of type *optdata_dn*. The data is read by the task issuing the GETSOCKOPT call with this option.
- **DSO_FLOWCTRL** allows the DECnet Transport protocol (NSP) to use flow control techniques. Flow control can have a major effect on application performance, and is dependent on how the application uses the network and the basic performance of the systems involved. Two settings are available for DSO_FLOWCTRL: NoFlow or Segment Count. The following values apply to each setting:

0 = NoFlow (also known as XON/XOFF or SEND/DONTSEND)

1 = Segment Count (this is the default)

DSO_FLOWCTRL can only be set before a socket has been connected, or, if receiving a connect, before the connect has been accepted. If you set this option on a listening socket, it will be passed on to new connects. An EBUSY error is returned if the socket is in the wrong state. There is no *getsockopt()* support for this option.

The DNP /FC: switch sets the default flow control for all NETBIOS sockets. NETBIOS sockets do not have a time period in which options can be adjusted. The PCSA MS-NET File Services protocol (SMB) is also a request-response protocol, and performs better in most cases with the /FC: switch set to zero (/FC:0).

Segment Count is the default DECnet-DOS technique and works well under adverse conditions. NoFlow works well with request-response protocols or on systems with robust network devices.

Under Segment Count, the receiver sends a count of segments that the transmitter is allowed to send. More credits are sent only when the application receives these segments. Each time credits are sent, the process involves sending another NSP link service message which also must be acknowledged by the other system. Due to restrictions in implementation, DECnet-VAX is incapable of carrying the link service acknowledgement on top of the normal data channel, and generates a separate link service acknowledgement message.

SETSOCKOPT

Under NoFlow, the link starts implicitly in a Send state. The transmitter is allowed to transmit until the receiver sends a link service message with a DontSend bit set. In this case, the receiver is responsible for keeping up with the transmitter. If the receiver is getting behind, it is then responsible for sending an "XOFF" message. If the receiver is poorly matched to the transmitter or if it has Ethernet hardware problems, excessive XON/XOFF messaging and retransmission will cause this situation to be worse than it would be under Segment count.

However, if the protocol is request-response, the receive pipeline can be adjusted to receive all of a transmitted response and no flow control messages need be exchanged at all. NoFlow is the default type used by DECnet-VAX.

Additionally, if the sending and receiving systems have well-matched performance (that is, data can be processed as fast as it is being sent), the No-Flow option allows data to be exchanged with lower overhead than the Segment count option.

- **DSO_LINKHOLD** allows the user to specify a time period in seconds during which the data retry algorithms continue to attempt to recover the link. This is in addition to the usual NSP retries, which are primarily controlled by the EXECUTOR RETRANSMIT FACTOR parameter. After this period, the link is disconnected with the message ENODEUNREACH. Use this value to override the global value set by the Network Control Program (NCP) for the EXECUTOR confidence timer. This value can range from 1 to 32767.

The default for this option is specified by the EXECUTOR CONFIDENCE TIMER parameter (currently 15 seconds). This timeout does not affect either link loss due to local resource failure or operation of the outgoing or incoming connection timers.

Input Data

Header Fields:

io_fcode = 25 (SETSOCKOPT function code).

io_socket the number for a socket created on the ACCEPT and/or deferred mode ACCEPT or ATTACH call.

io_flags set as follows:

MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.

io_psize specifies the size of the data structure *sockopt_dn* as 8 bytes.

Parameter Fields:

sockopt_dn specifies the socket option data structure.

The structure contains the following data fields:

sop_level 2 bytes; specifies the level at which options are manipulated.

If the level is set to SOL_SOCKET, then *sop_optval* and *sop_optlen* are ignored. If the level is set to DPROTO_NSP, then the rest of the data structure can contain either access control or optional user data, or access mode information.

sop_optname 2 bytes; specifies options to be interpreted.

When the socket level is set to DPROTO_NSP, *sop_optname* can be set to one of 6 specific options (for example, DSO_CONDATA).

SETSOCKOPT

sop_optval 4 bytes; specifies access option values used with the *sop_optlen* SETSOCKOPT and the GETSOCKOPT calls.

The interpretation of each argument is function-dependent as shown here:

sop_optval specifies the pointer to a buffer which contains information for setting access option values.

sop_optlen specifies the size of the option value buffer.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SETSOCK OPT call:

[EACCES] unable to disconnect the socket.

[EBADF] the argument *io_socket* does not contain a valid descriptor.

[ECONNABORTED] the client task disconnected before the ACCEPT call completed.

[EDOM] the acceptance mode is not valid.

[EFAULT] system detected an invalid buffer.

[ENOBUFS] no buffer space available. Socket cannot be created. No more logical links.

SETSOCKOPT

[ENOPROTOOPT]	there was no access control information supplied with the connection request.
[EOPNOTSUPP]	the option is unknown.
[ENETDOWN]	the local network is OFF.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.
[E2BIG]	<i>io_psize</i> is too large.

SHUTDOWN

SHUTDOWN

SHUTDOWN — shut down all or part of a full duplex logical link.

DESCRIPTION

Use the SHUTDOWN call to indicate that messages will no longer be sent or received on the socket.

Input Data

Header Fields

io_fcode = 7 (SHUTDOWN function code).

io_socket the number for a socket created on the ACCEPT or ATTACH call.

io_flags set as follows:

MSG_ASYNC (0x0008) for asynchronous mode, optional.

MSG_CALLBACK (0x0010) to specify a callback, optional.

MSG_NIOCB (0x0080) must be set.

MSG_USRWAIT (0x0200) to specify a user wait loop, optional.

io_psize specifies the size of the required data structure *shutdown_dn* as 2 bytes.

Parameter Fields:

shutdown_dn specifies the type of shutdown. This structure contains the following data field:

snd_how specifies the type of shutdown. This argument can be set to:

0 which disallows further receives.

1 which disallows further sends.

SHUTDOWN

2 which disallows further sends and receives.

Output Data

Header Fields

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SHUT DOWN call:

[EBADF]	the argument <i>io_socket</i> does not contain a valid descriptor.
[ENOTCONN]	the specified socket is not connected.
[ENETDOWN]	the local network is OFF.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.
[E2BIG]	<i>io_psize</i> is too large.

SIOCTL

SIOCTL

SIOCTL — control the operations of open sockets.

DESCRIPTION

Use the SIOCTL call to control the operation of open sockets. The call indicates whether an argument is an input or output argument and indicates the size of the specific argument in bytes.

Input Data

Header Fields:

io_fcode = 24 (SIOCTL function code).

io_socket specifies the socket number.

io_flags set as follows:

MSG_ASYNC (0x0008) for asynchronous mode, optional.

MSG_CALLBACK (0x0010) to specify a callback, optional.

MSG_NIOCB (0x0080) must be set.

MSG_USRWAIT (0x0200) to specify a user wait loop, optional.

io_psize specifies the size of the data structure *sioctl_dn* as 8 bytes.

Parameter Fields:

sioctl_dn specifies the socket I/O control function data structure. The structure contains the following data fields:

sio_s is ignored. Must be zero.

sio_request specifies the I/O control function to be used. The control levels include:

FIONREAD	returns the total byte count of all messages waiting to be read. The <i>argp</i> field points to a word.
FIONBIO	sets/clears blocking or nonblocking I/O operation. The <i>argp</i> field points to a byte that contains a value of 0 or 1. For blocking I/O, <i>argp</i> should point to a value of 0. For nonblocking I/O, <i>argp</i> should point to a value of 1.
FIORENUM	renumbers an assigned socket number to another number. In this way, the original socket number is made available again. The <i>argp</i> field points to a word.
The SELECT	function call cannot accept socket numbers that exceed 1 to 34 range. (Refer to the SELECT call description in this chapter for details.) If you specify a socket number that is already in use, an error message, EEXIST, is returned.

argp specifies the address of the argument list.

Output Data

Header Fields:

io_status completion status field:

0 = call completed successfully

-1 = error completing call

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SIOCTL call:

[EBADF] the argument *io_socket* does not contain a valid socket number.

SIOCTL

[EOPNOTSUPP]	the socket type does not support the socket I/O operation.
[ENETDOWN]	the local network is OFF.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.
[E2BIG]	<i>io_psize</i> is too large.

SOCKADDR

SOCKADDR — retrieve local socket information that is set by the BIND call.

DESCRIPTION

The SOCKADDR call returns socket information set by the BIND call. If no BIND call was ever executed, undefined results are returned in output data fields.

Input Data

Header Fields:

<i>io_fcode</i> = 15	(SOCKADDR function code).
<i>io_socket</i>	specifies the number for a socket which was bound to a name on the BIND call.
<i>io_flags</i>	set as follows:
MSG_ASYNC (0x0008)	for asynchronous mode, optional.
MSG_CALLBACK (0x0010)	to specify a callback, optional.
MSG_NIOCB (0x0080)	must be set.
MSG_USRWAIT (0x0200)	to specify a user wait loop, optional.
<i>io_psize</i>	specifies the size of the data structure <i>sockaddr_dn</i> as 26 bytes.

Output Data

Header Fields:

<i>io_status</i>	completion status field:
0	= call completed successfully
-1	= error completing call

SOCKADDR

-2 = call pending (asynchronous mode only)

io_errno contains additional error detail. Errors returned for the SOCKADDR call:

[EBADF]	the argument <i>io_socket</i> does not contain a valid socket number.
[ENETDOWN]	the local network is OFF.
[ENOBUFS]	no buffer space available. Socket cannot be created. No more logical links.
[E2BIG]	<i>io_psize</i> is too large.

Parameter Fields:

sockaddr_dn specifies the socket address data structure. A user retrieves data from the fields filled in by this function call.

The following data fields can be filled in by this function call:

<i>sdn_family</i>	specifies the address family AF_DECnet.
<i>sdn_objnum</i>	the object number for the local task. It can be a number from 0 to 255.
<i>sdn_objname1</i>	is the size of the object name.
<i>sdn_objname</i>	is the object name of the local task. It can be up to a 16-byte array. It is only used when <i>sdn_objnum</i> equals 0.
<i>sdn_add</i>	specifies the address data structure that contains the local node name and address.

NETBIOS Emulation Interface

The NETBIOS emulation interface, developed by Digital Equipment Corporation, emulates the industry-compatible NETBIOS interface. The DECnet-DOSTM NETBIOS emulation interface lets industry-compatible NETBIOS applications communicate in a transparent manner using DECnetTM as a transport mechanism. With the NETBIOS emulation interface installed on your PC, your NETBIOS applications can communicate with other NETBIOS applications or DECnet applications in a DECnet network. The NETBIOS emulation does not provide the ability to communicate with other vendors' proprietary protocols or network data links.

This chapter presents a general overview of Digital's NETBIOS emulation interface and describes the calls that your applications can use to access this interface. Note that this chapter uses the terms "NETBIOS emulation," "NETBIOS emulator," and "NETBIOS" when referring to the Digital NETBIOS emulation interface.

7.1 Introduction

NETBIOS is a session-layer interface for network applications that are written to use the IBM® PC Network Program, the IBM LAN Support Program, or other industry-compatible network products.

Digital's NETBIOS emulation supports network sessions by mapping NETBIOS functions into DECnet protocols. This mapping lets you communicate easily with other DECnet systems because the other systems do not have to know that your application is using the NETBIOS interface.

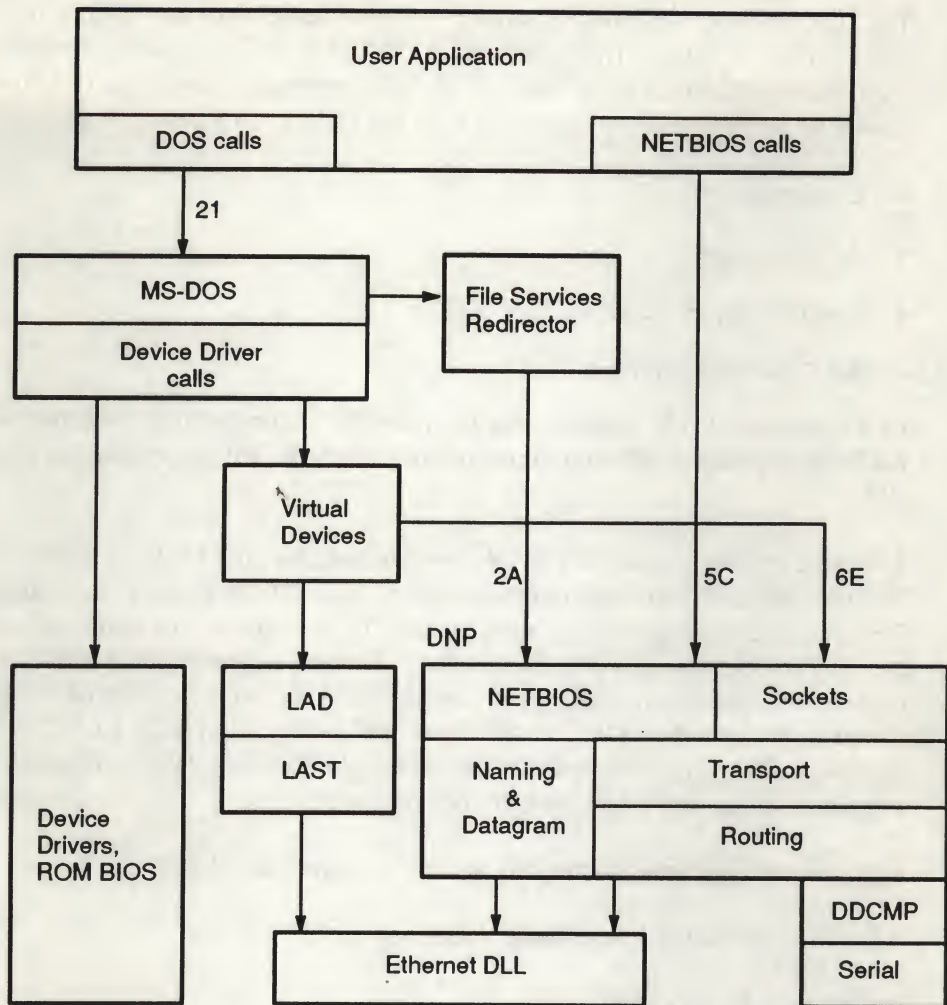
The NETBIOS emulation interface provides support for:

- Local area network (LAN) communications
- Wide area network (WAN) communications
- Communication to VAX[™] computers (or other computers supporting DECnet)

In every case, the NETBIOS emulation software module handles all network communication for your application, including the following:

- Initiation of a communications link
- Exchange of messages
- Termination of the communications link

The following diagram illustrates how NETBIOS emulation maps to other network processes.



7.1.1 LAN Communications

The DECnet-DOS NETBIOS interface emulates the services provided by the IBM LAN Support Program. These services let PC NETBIOS applications communicate with other NETBIOS applications in an Ethernet local area network (LAN). The following services are implemented by the NETBIOS LAN protocol:

- Datagram services
- Adapter name
- Remote adapter status

7.1.2 WAN Communications

In addition to LAN support, Digital provides extended functions that allow NETBIOS applications to communicate with other NETBIOS applications across a WAN.

To enable communication in a WAN environment, the NETBIOS emulation interface provides name services for the resolution of NETBIOS names to corresponding DECnet node address and node name pairs. To accomplish this name resolution, the NETBIOS emulator maintains a volatile database of predefined names that map to DECnet addresses, objects, and names. The DECnet node information that is stored in the permanent DECnet databases (DECNODE.DAT, DECREM.DAT, and DECALIAS.DAT) is automatically loaded into the volatile database. Datagram and adapter status services, however, are not supported.

Note that the following commands are not supported in a WAN:

ADAPTER STATUS (for Remote Names)
FIND NAME
RECEIVE BROADCAST
RECEIVE DATAGRAM
SEND BROADCAST DATAGRAM
SEND DATAGRAM

7.1.3 Communication to VAX Computers

Extended functions also allow NETBIOS applications to communicate with a computer other than a PC running the Digital NETBIOS emulation if the target computer is running DECnet software. NETBIOS allows communication to any system that is running DECnet software by mapping session support calls directly to DECnet-DOS Network Process (DNP) calls.

To enable communication to VAX computers, the NETBIOS emulation interface provides a means to map names to corresponding DECnet node address and node name pairs. To accomplish this name resolution, the NETBIOS emulator maintains a volatile database of predefined names that map to DECnet addresses, objects, and names. The DECnet node information that is stored in the permanent DECnet databases (DECNODE.DAT, DECREM.DAT, and DECALIAS.DAT) is automatically loaded into the volatile database.

7.2 How Applications Communicate

NETBIOS applications communicate on the network by using names. A local name refers to a NETBIOS application running on your PC. Each NETBIOS name refers to an application running on a remote computer.

In DECnet terms, a NETBIOS name can be regarded as a network service name. The NETBIOS name identifies the local program for incoming messages or sessions, and specifies the target of outgoing messages or sessions.

Digital's NETBIOS emulation interface implements a naming service that provides for the following:

- Registration and resolution of names between PC nodes running NETBIOS emulation in an Ethernet LAN environment.
- Resolution of remote adapter names to DECnet address and names for communication in a WAN environment, or for communication to a DECnet node that is not running NETBIOS.

NETBIOS maintains a table of local names. To use the network, you must follow these steps:

1. Add the application's name to the local name table in order to identify it to the network.
2. Establish a communications session with a remote name on the network.
3. Send and receive messages using that communications session.

When NETBIOS receives a request from an application to communicate with another application identified by a name, it checks the volatile database to see if that name is predefined. If the name is predefined, NETBIOS uses the DECnet address/object name/number to establish communication. If the name is not predefined, the NETBIOS naming service maps the name to the node address. Note that communication to a node that has not been predefined is supported in a LAN environment only. An adapter name can be mapped to a fully qualified DECnet network address having the following components:

Address	The DECnet address of the target node.
Object	An object definition, either numbered or named, to which the connection is being established.
Access control	An optional access control string to present to the target node.

Names are defined as follows:

- Dynamically — on an Ethernet LAN by PCs running NETBIOS or by VMS systems supporting NETBIOS naming services.
- By management — using the NCP commands SET (or DEFINE) LOCAL (or REMOTE) and DEFINE NODE *number* MS-NET.
- By program — using the DEC Extended Commands (DSCB interface).

7.3 NETBIOS Command Line Switches

DECnet-DOS provides the NETBIOS emulation software as an optional variation of the DECnet Network Process (DNP). Users can select the variation while installing DECnet-DOS. For more information on DNP variant file names, refer to the manual entitled *Installing DECnet PCSA Client for DOS (with Diskettes)*.

You can load DNP from a batch file (such as AUTOEXEC.BAT) or by typing DNP at the DOS prompt.

Once you invoke DNP, a subsequent try will not cause a second copy of the software to be loaded into memory. You may be able to unload DNP if you are using conventional memory. If you do so, the NETBIOS emulation is unloaded with it. If you are using Expanded Memory Services, you will not be able to unload just DNP from memory; however, you will be able to unload the whole network using an unload batch file.

Table 7-1 provides the optional command switches you can set when invoking the DNP with NETBIOS emulator software. These switches provide parameters to NETBIOS when used on the DNP command line after the default database path.

Format for DNP switches:

DNP [<decnet-db-path>] [/<switch>:<value>] [/<switch>:<value>] ...

Example for using the DNP switches /NAM and /REM:

```
DNP C:\DECNET\ /NAM:Y /REM:4
```

Note that DNP parses only the first two characters of a switch. Invalid switches or values outside of the allowed range will cause an error, and the parsing will continue with the next switch.

In this chapter, command code values are represented in hexadecimal units.

Table 7-1: Command Line Switches for DNP with NETBIOS Emulation

Command Switch	Description	Default Value
/CMD: <i>n</i>	Number of additional simultaneous commands supported	# of SDBs
/FC: <i>n</i>	Flow control option for NETBIOS links (0=NoFlow,1=Segment count)	Segment
/I2A:[Y N]	Support NCBs on Interrupt 2Ah	Yes
/LAN: <i>n</i>	LAN adapter number supported by the emulator	0
/LCN: <i>n</i>	Number of local names (min=2, max=64)	16
/M:[E D]	Multicast listening (ignored - follows setting of Circuit Multicast Listen)	Enabled
/MSN: <i>n</i>	Number of server names (min=1, max=64)	# MS-NET nodes defined
/NAM:[Y N]	Add node name to local name table as Server and Client automatically	Yes
/NBS: <i>n</i>	Largest session message size supported (min=256, max=65535)	65535
/REM: <i>n</i>	Number of remote names (min=1, max=64)	1 or # defined in DECRET.DAT (also 1 if no names defined)

7.4 Accessing NETBIOS Services

Your application program sends data to NETBIOS by issuing a series of function calls. These function calls provide the information needed to establish a communications session with a target application on a remote node.

To issue function calls to NETBIOS, your application must build a special network data structure (NCB) for each call, then issue an interrupt to vector 5C hex (INT 5CH) or 2A hex (INT 2AH). NETBIOS uses this data structure to exchange information with your application program.

You can use either INT 2A or INT 5CH with the DECnet-DOS NETBIOS emulator (they are equivalent). In other NETBIOS implementations, INT 5C is usually the direct interface to the network device driver, while INT 2A provides additional services (such as error retry). Error retry is part of the DECnet-DOS emulation.

7.4.1 NETBIOS Installation Checks

Before it issues any calls to NETBIOS, your application should perform an installation check to see if the NETBIOS emulation software is installed.

7.4.1.1 Checking 2A Installation

Issue an interrupt 2A to determine if the NETBIOS handler is installed:

Entry	Register Contents
AH	00H – Installation check

Exit	Register Contents
AH	Installation Flag = 0 interface is not installed = Nonzero, interface is installed

Use this function before issuing any other interrupt 2AH function request. When AH returns with a zero value, the interface is **not** installed. Therefore, no other interrupt 2AH function requests should be issued because meaningless data may be returned by such calls.

7.4.1.2 Checking 5CH Installation

Your application should issue a DOS function request to read the contents of the 5CH interrupt vector. If the location contains binary zeroes, the NETBIOS interface is not installed. If the location contains any value other than zero, issue either command 7FH or FFH. NETBIOS will return a code of 03H to the AL register indicating that the NETBIOS is present.

The following illustrates the INT 5C installation check:

Entry	Register Contents
ES:BX	Pointer to the Network Control Block NCB_COMMAND = 7FH

Exit	Register Contents
AH	Installation Flag = 0 interface is not installed = Nonzero, interface is installed
AL	= 03H Return Code

7.4.2 Issuing NETBIOS Calls

An application must complete the following four-step process when issuing a call to NETBIOS:

1. Create Network Control Block (NCB)
 - Build and fill required NCB fields
 - Allocate any necessary buffers specified in the NCB fields
2. Set a pointer to the address of the NCB (in ES:BX)
3. Set AX = 0400 (if using INT 2A)
4. Issue the software interrupt to vector 5C hex (INT 5CH) or vector 2A hex (INT 2AH)

After NETBIOS processes a command, it returns control to your application. NETBIOS returns status to your program in the form of return codes to the NCB and to register AX (NCB_RETCODE and NCB_CMPCPLT).

On Return	Return Codes
AH	NCB Return Code = 0 – success = 1 – error
AL	= 0 – success := 0 – (nonzero) error code

7.5 The Network Control Block (NCB)

You use NCB commands to communicate with the NETBIOS emulator. However, before your application can issue an NCB command, it must construct a data structure that the NETBIOS emulator can use to exchange data with your application. This data structure is called a Network Control Block (NCB).

Your application supplies values for the various NCB fields on each NCB call. The NCB must contain all the information required to perform the function you request. Table 7-2 summarizes the NCB fields.

Table 7-2: Network Control Block Fields

Field Name	Size (Bytes)	Description
NCB_COMMAND	1	Command code field. You can issue NCB commands in asynchronous or synchronous mode. Set the high-order bit to indicate an asynchronous operation.
NCB_RETCODE	1	Return code field. NETBIOS returns the status of the requested operation to this field as follows: FFH — Command pending 00H — Command completed All other values indicate an error condition.
NCB_LSN	1	Local session number field. Returned by the CALL or LISTEN commands and used to identify the logical link on subsequent requests.
NCB_NUM	1	Identifies a local name. Returned by the ADD NAME and ADD GROUP NAME commands. NETBIOS returns a value from 2 to 254 in this field.
NCB_BUFFER	4	Pointer to message buffer address in segment:offset form.
NCB_LENGTH	2	Buffer length field. NETBIOS updates this field to indicate the number of bytes actually received. This field specifies the size of the message to transmit or the length of the buffer for receiving messages. In the output field, this number is updated to reflect the number of bytes actually received.

Table 7-2 (Cont.): Network Control Block Fields

Field Name	Size (Bytes)	Description
NCB_CALLNAME	16	Indicates the name with which you want to communicate. For Chain Send, the first 2 bytes indicate length of second buffer. Next 4 bytes indicate the second buffer address.
NCB_NAME	16	Indicates the name in local adapter name table that your application is known by on the network. Must use all 16 bytes.
NCB_RTO	1	Receive timeout value, in 500-millisecond intervals, for a logical link. Receive timeout is the maximum amount of time allowed before a RECEIVE command returns a time-out error. A zero value in this field indicates no timeout.
NCB_STO	1	Send timeout value, in 500-millisecond intervals, for a logical link. Not used. Use the NCP commands SET EXECUTOR DELAY FACTOR and SET EXECUTOR RETRANSMIT FACTOR to change timeout values. Refer to the <i>DECnet-DOS Network Management Guide</i> for help with NCP commands.
NCB_POST	4	Pointer to post routine in segment:offset form. This field, used in asynchronous operations only, points to a callback routine to be executed after the command has been completed. A zero value with an asynchronous call indicates that no callback will be called. Your application can check the NCB_CMD_CPLT field for completion status. NETBIOS returns a value to the NCB_CMD_CPLT field as follows: FFH - Command pending non 0xFFH - Command completed
NCB_LANA_NUM	1	Adapter number that this command is directed to. The primary adapter is zero.

Table 7-2 (Cont.): Network Control Block Fields

Field Name	Size (Bytes)	Description
NCB_CMD_CPLT	1	Command status field. NETBIOS returns status as follows: FFH — Command pending 00H — Command successful All other values indicate an error condition.
NCB_RESERVE	14	Reserved field. Must be present.

7.6 Network Control Block (NCB) Commands

The NCB commands let you send data to NETBIOS. These commands are divided into four categories: general, name, session, and datagram. Each category has a class of commands that provide the following types of support:

- **General** — Used to control NETBIOS or to get status information about a NETBIOS process.
- **Name** — Allows your network applications to be known by a unique name on the network. NETBIOS checks to see if a name you supply with the ADD NAME command is a unique name. The NETBIOS emulator returns an error message if anyone else on the network is using the same name.

The local node name table can hold up to 16 names. You can delete all names from the local name table with the RESET command. This command removes all the names from the table with the exception of two reserved names.

- **Session** — Establishes a logical link between any two names on the network. These names can be listed in your local name table or in any other NETBIOS name table. The session support commands allow you to establish a logical link, to send and receive data, to obtain status, and to drop logical links. Many commands can be outstanding, limited by the system defined maximum.

NETBIOS uses a 1-byte number to refer to each session after it is established. NETBIOS returns this number to the application in the NCB_LSN field of the NCB after the session is established.

Each name pair can support multiple active sessions. The maximum number of active sessions depends upon how you have configured your system. For example, if you have used NCP to set your maximum links to 4, then you can establish only 4 active sessions at one time. If you set maximum links to 32, then you can establish up to 32 active sessions at one time.

NETBIOS supports session sizes ranging from 0 to 65,535 bytes.

- **Datagram** — Allows your applications to send messages to a name or a group name or to multicast (broadcast) messages to all systems running the NETBIOS emulator interface on the Ethernet LAN. These commands also enable you to receive Datagram messages from a specific name or group name, or from anyone on the LAN.

The message you send or receive can be from 0 to 512 bytes in length.

NOTE

Datagram support is only implemented between nodes running the NETBIOS emulation interface in an Ethernet LAN.

7.6.1 Using Asynchronous and Synchronous Modes

You can issue most commands to the NETBIOS emulator in either synchronous or asynchronous mode. The RESET and CANCEL commands are exceptions and can be issued in synchronous mode only.

If you issue a command in synchronous mode, your application program waits (blocks) until the command is complete. If you issue a command in the asynchronous mode, your application program continues to execute while the command is completely processed by NETBIOS. Use the mode most appropriate for your application.

- **Asynchronous mode**

If you issue commands in the asynchronous mode, you can code your application to check for completion by either:

- Specifying a callback routine that will be executed when the command completes

or

- Polling the NCB_CMD_CPLT field of the NCB for completion

If you use polling, your program must check for a change in value in the NCB_CMD_CPLT field. This field will change from command pending (FFH) to the final completion code (non-FFH).

If you issue an asynchronous form of a command, the NETBIOS returns an immediate value to the AL register that indicates whether the command has been accepted or rejected.

The NETBIOS emulator may respond with these immediate return codes:

- 00H - Command accepted
- 03H - Invalid command
- 21H - Interface busy
- 22H - Too many commands; all NCBs are in use
- 23H - Invalid number in NCB_LANA_NUM
- 24H - Command completed during cancel
- 26H - The requested command is invalid to cancel
- 4XH - Network error

If you issue an asynchronous command and specify a callback routine, NETBIOS places this value in the NCB_RETCODE field. If you do not specify a callback routine, NETBIOS places the final value in the NCB_CMD_CPLT field. Your application should not poll the NCB_RETCODE field to determine if an asynchronous command completed.

When a callback routine is executed, the AL register will also contain the completion status of the command.

- Synchronous mode

If you issue commands in synchronous mode, the NETBIOS emulator returns status to your application in the NCB_RETCODE field.

Table 7-3 summarizes the NCB commands you use to pass data to the NETBIOS emulation interface.

Table 7-3: NETBIOS Emulator Commands

Command Name	Description
General Commands	
ADAPTER STATUS	Obtain adapter status of local or remote node
CANCEL	Cancel previously issued command
RESET	Reset local adapter; clear name and session tables
Name Support	
ADD GROUP NAME	Add group name to local adapter name table
ADD NAME	Add a name to local adapter name table
DELETE NAME	Delete name from local adapter name table
FIND NAME	Find the location of an adapter name on the network
Session Support	
CALL	Create a logical link with local or remote node
CHAIN SEND	Send two buffers together on the logical link
HANG UP	Close the logical link
LISTEN	Listen for connect request from a local or remote node
RECEIVE	Receive data on a logical link
RECEIVE ANY	Receive data from any logical link
SEND	Send data on the logical link
SESSION STATUS	Receive status on all active sessions
Datagram Support	
RECEIVE BROADCAST DATAGRAM	Receive broadcast Datagram from any name
RECEIVE DATAGRAM	Receive Datagram message
SEND BROADCAST DATAGRAM	Send broadcast message to any name or group name
SEND DATAGRAM	Send Datagram message to name or group name

7.6.2 Return Codes for NCB Commands

Because the NETBIOS emulator commands are now part of the DECnet-DOS Network Process (DNP), your NETBIOS application may receive DECnet return codes from DNP as well as from NETBIOS. NETBIOS returns these codes in the following format:

80H + *nn*H, where *nn*H represents a DECnet error code

7.7 Accessing Digital Extended Functions

Digital provides extended functions to the standard NETBIOS interface that allow your NETBIOS applications to communicate with non-NETBIOS nodes in a DECnet network. These extended functions enable support for NETBIOS names and their resolution to DECnet node numbers and names.

You present commands to access these extended functions in the same way that you present NCB commands. That is, your application must construct a data structure that the NETBIOS emulation interface can use to exchange information with your application. This data structure is called a Digital Session Control Block (DSCB).

Your application supplies values for the various fields for each DSCB function call. The DSCB must contain all the information required to perform the function you request. After it builds a DSCB, your application can issue a call to the NETBIOS interface, using INT 2AH only.

The NETBIOS interface you access through INT 2A provides two distinct sets of services. The first is the industry-compatible functions specified with AH = 04H as described earlier in this chapter. The second is a set of Digital's extended functions. These extended functions enable support for communication to DECnet nodes.

NETBIOS maintains two tables of names that enable the resolution of adapter names to DECnet node names and addresses. Your application uses the DSCB commands to add DECnet names to or remove DECnet names from these tables.

Functions 0 through 7 reference NETBIOS names to use for connection to MS-Networks Servers. These servers use the Microsoft® SMB protocol. Digital uses DECnet object number 64 for these connections.

Functions 21 through 26 reference NETBIOS names to use for connection to any DECnet application. The user can specify the desired DECnet connect information in these entries.

The following illustrates how to issue an INT 2AH function call to access the Digital-specific functions.

Entry	Register Contents
AH	0DCH
ES:BX	Points to address of DEC-SES Control Block

Exit	Register Contents
AH	Error Flag 00H no error occurred (AL = 00H) 01H error occurred (AL = error)
AL	Return Code

Table 7-4 summarizes the DSCB fields.

Table 7-4: Digital Session Control Block (DSCB) Fields

Field Name	Size (Bytes)	Description
DSCB_CMD	1	Command code field
DSCB_ERR	1	Return code field
DSCB_INDEX	1	Index number of the node entry; used only for DSCB functions 0-7
DSCB_NAME	16	Specifies the name of the adapter/node name
DSCB_NUM	2	Specifies the DECnet node address, in 16 bit format as follows: Bits 31 - 26 = area number Bits 25 - 0 = node number
DSCB_OBJTYPE	1	Indicates DECnet object type of remote adapter
DSCB_ACC_LEN	2	Length of account name if providing optional access control data
DSCB_ACC	40	Indicates account name, zero terminated if providing optional access control data
DSCB_PASS_LEN	2	Indicates the password length if providing optional access control data
DSCB_PASS	40	Specifies password, zero terminated if providing optional access control data
DSCB_USER_LEN	2	Indicates the user name length
DSCB_USER	40	Specifies the user name, zero termi- nated

Table 7-5 lists the DSCB commands.

Table 7-5: DEC Extended Function Commands

Command Name	Function Code (Hex)	Description
INSTALLATION CHECK	00	Use to check if NETBIOS emulation with the DEC extensions is installed; issue this before issuing any other Digital extended functions
ADD SERVER NAME	01	Use to add a server to remote server table
DELETE SERVER GIVEN NODE NUMBER	02	Delete server in the remote server table by node number
DELETE SERVER NODE NAME	03	Delete server in the remote server GIVEN table by node name
READ SERVER GIVEN NUMBER	04	Read server in the remote server NODE table by node number
READ SERVER GIVEN NAME	05	Read server in the remote server NODE table by node name
READ SERVER BY	06	Read a server in the remote server INDEX table
DELETE ALL SERVER NAMES	07	Delete all entries in the remote server table
ADD REMOTE NAME	21	Add remote name definition with optional access control data
DELETE REMOTE NAME GIVEN NAME	22	Remove remote adapter name by name
READ REMOTE NAME	23	Read the remote name definition by name
READ NUMBER OF REMOTE NAMES	24	Get the number of entries in the table of remote adapter names
READ ALL REMOTE	25	Read all the remote names in the NAMES remote adapter name table
DELETE ALL REMOTE NAMES	26	Clear all remote adapter name entries

The DSCB interface may return the following codes to your application:

- 00H - Function completed successfully
- 01H - Table full, cannot add another entry
- 02H - Duplicate name, node name is currently used by another entry
- 03H - Duplicate number, node number is currently used by another entry
- 04H - Node name not found
- 05H - Node number not found
- 06H - No entry found with given index
- 07H - Index given is out of range
- 08H - Illegal function number
- 09H - Out of resource, no stack space currently available to perform the function, try again later
- 0AH - Cannot delete own node entry
- 0BH - Buffer too small, cannot fill data into buffer area
- 0CH - No table entries configured

7.8 NCB and DSCB Commands

The remainder of this chapter describes the NCB commands (in alphabetical order) followed by the DSCB commands (in alphabetical order). Each description provides a summary of the fields required for input and output as well as the final return codes that may be returned to your application.

7.8.1 NCB Commands

The following are the NCB commands.

ADAPTER STATUS

Use this command to obtain the status of a local or remote NETBIOS. Applications can request status by specifying a name which is local to the adapter in the NCB_CALLNAME field. If you specify an asterisk (*) in the first byte of NCB_CALLNAME field, NETBIOS returns the local information.

NETBIOS returns information described below in the address you specify in the NCB_BUFFER field. The minimum number of bytes returned is 60. The maximum number will depend on the number of names defined in the adapter name table. The maximum size of the status buffer is 348 bytes. Its actual size is $60+18*n$ where n is the number of names defined in the adapter name table.

NETBIOS returns the following data to this buffer:

Ethernet address	6 bytes, contains the DECnet Ethernet address assigned to this node.
External jumper status	1 byte, returns zero in current implementation.
Power-on self-test	1 byte, returns an 80H in current implementation.
Software version	2 bytes, indicates emulator version number and revision level. Byte 0 : Version number Byte 1 : Revision level
Statistics:	
Traffic and error	2 bytes, not used. 2 bytes, receive failures, number of CRC errors, and all framing errors. 2 bytes, data overrun errors. Will not increment further when count equals 0FFFFH. 2 bytes, collision detect failures. This counter rolls over in PC-NET, but not in DECnet. Supported in Datalink. 2 bytes, send failures. This counter rolls over on PC-NET, but not in DECnet. Supported in Datalink.

ADAPTER STATUS

4 bytes, number of successfully sent packets. Value will roll over. Supported in Datalink.

4 bytes, number of successfully received packets. Value will roll over. Supported in Datalink.

2 bytes, number of retransmissions. Value will roll over. Not supported in Datalink directly. What will be supplied in this field will be the Datalink's Initially Deferred count.

2 bytes, number of system buffer allocation failures. After the counter reaches a value of 0FFFFH, does not increment further.

Adapter resource

8 bytes, reserved for internal use.

2 bytes, number of free command blocks.

2 bytes, maximum number of NCBs defined.

2 bytes, maximum number of free command blocks.

4 bytes, reserved.

2 bytes, number of pending sessions. The value returned will be the number of LISTEN-pending plus the number of CALL-pending plus established sessions, sessions aborted, hang-up pending or hang-ups.

2 bytes, maximum possible pending sessions. The value returned will always be either 32 or the maximum number of sessions that the NETBIOS is currently configured for, whichever is less.

2 bytes, maximum possible sessions. The value returned will always be either 32 or the maximum number of sessions, whichever is less.

2 bytes, maximum data packet size.

Local name table

16 entries of 18 bytes. The first 16 bytes of each entry are the name, the last two bytes are status information. The first status byte gives the name number. The second byte when masked with 87H will return the following values:

NXXXX000 - Registering a name

NXXXX100 - A registered name

NXXXX101 - A deregistered name

NXXXX110 - A duplicate name

NXXXX111 - A duplicate name, de-register pending

ADAPTER STATUS

where

X = Reserved.

N = 0 The name is unique.

N = 1 The name is a group name.

If the name for which you are requesting status is remote, the remote system will execute an ADAPTER STATUS function and return the information in a datagram message to the requester. The remote system must respond within a specified time-out period; otherwise error code 05H (command timed out) is returned.

NOTE

ADAPTER STATUS is supported only between nodes that are running NETBIOS on a LAN.

Command code:

33H - Synchronous request.

B3H - Asynchronous request.

Input fields:

NCB_BUFFER	the address of the status.
NCB_LENGTH	the length of the status buffer supplied to the call.
NCB_CALLNAME	is either a remote name or an * for local status.
NCB_POST	address of the callback routine.
NCB_LANA_NUM	

Output fields:

NCB_LENGTH updated to reflect the amount of status information loaded into the buffer.

NCB_RETCODE

ADAPTER STATUS

Final return codes:

- 00H - Success
- 03H - Invalid command
- 01H - Illegal buffer length
- 05H - Command timed out
- 06H - Buffer too small
- 0BH - Command canceled
- 19H - Duplicate name found
- 21H - Interface busy
- 22H - Maximum number of outstanding commands
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

ADD GROUP NAME

Use this command to add a 16-character group name to the local name table. The name must not be in use as a unique name by any other node on the LAN, but can be used by other nodes as a group name.

A check is made to ensure that the name does not begin with either an asterisk (*) or 00H. The local name table is then searched to determine that the name is not already present, as a unique name. The name is then added to the table and the status flagged as "register in progress." A multicast message is sent requesting that any remote node respond if the name is already registered as a unique name in another adapter name table. If no response is received within a predefined timeout period, the multicast message is repeated once.

If a response is received, the name is removed from the local name table and the command rejected; otherwise, the command is accepted and the name marked as "registered."

Command code:

36H - Synchronous request.

B6H - Asynchronous request.

Input fields:

NCB_NAME the 16-byte name being registered.

NCB_POST the address of the callback routine if the asynchronous request is used.

NCB_LANA_NUM

Output fields:

NCB_NUM used to reference the name for Datagram commands.

NCB_RETCODE

NCB_CMD_CPLT

ADD GROUP NAME

Final return codes:

- 00H - Success
- 03H - Invalid command
- 0DH - Name already in local name table
- 0EH - Local name table full
- 15H - Name begins with * or 00H
- 16H - Name being used by remote system
- 19H - Name conflict detected
- 21H - Interface busy
- 22H - Too many commands, all NCBs are being used
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

ADD NAME

Use this command to add a name to the local name table. **You must add the name to the table before establishing a call or listening for a connect.** The node name must be a unique, 16-character name. The name must not be present in the local adapter name table or in the name table of any other node on the LAN.

NETBIOS checks to ensure that the name does not begin with either an asterisk (*) or 00H, then searches the local name table to determine that the name is not already present. If the name is a unique name, NETBIOS adds the name to the table and sets the status of the name to "register in progress."

While processing this command, NETBIOS sends a multicast message out on the network, requesting that any remote node respond if the adapter name is already registered in another name table.

If the NETBIOS emulator does not receive a reply within a predefined timeout period, NETBIOS assumes that the name is not being used elsewhere on the network. NETBIOS sets the status of the name to "registered." If NETBIOS does receive a response, it removes the name from the name table and returns the error message "name conflict detected."

NOTE

ADD NAME is supported only between nodes that are running NETBIOS on a LAN. If you are using NETBIOS in a WAN, ADD NAME returns success but it is not validated on the network,

Command code:

30H - Synchronous request.
B0H - Asynchronous request.

Input fields:

NCB_NAME the name to be added.

NCB_POST address of the callback routine if the asynchronous request is used.

ADD NAME

NCB_LANA_NUM

Output fields:

NCB_NUM used to reference the name you added to the name table for use with the Datagram and RECEIVE ANY commands.

NCB_RETCODE

NCB_CMD_CPLT

Final return codes:

- 00H - Success
- 03H - Invalid command
- 0DH - Name already in local name table
- 0EH - Local name table full
- 15H - Name begins with * or 00H
- 16H - Name being used by remote system
- 19H - Name conflict detected
- 21H - Interface busy
- 22H - Too many commands, all NCBs are being used
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

CALL

Use this command to create a logical link with either a local or remote name. You specify your local session name as established by an ADD NAME command. The local session name should appear in the NCB_NAME field and the destination name should appear in the NCB_CALLNAME field. The name you specify in NCB_CALLNAME must have a LISTEN command pending before the session can be established.

NETBIOS returns a value in the NCB_LSN field when the connect is complete. This value ranges from 1 to 254. NETBIOS uses this number for future references to this connection.

NETBIOS will abort RECEIVE commands for this logical link if you specify timeout intervals in the NCB_RTO field. You must specify timeout intervals in 500-millisecond units. If you specify zero in these timeout fields, no timeout will occur.

Digital's NETBIOS emulation does not support SEND timeouts. The connections made with the CALL command will end unsuccessfully after the outgoing executor timer expires. You must use the DECnet-DOS NCP commands SET EXECUTOR DELAY FACTOR and SET EXECUTOR RETRANSMIT FACTOR to change SEND timeout values. See the *DECnet-DOS Network Management Guide* for help with NCP commands.

Command code:

10H - Synchronous request.
90H - Asynchronous request.

Input fields:

NCB_CALLNAME the 16-character name being called.

NCB_NAME the local adapter name.

NCB_RTO the receive timeout to be used in 500-ms intervals, or zero if none.

CALL

NCB_STO not supported.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

NCB_LSN the logical link ID to be used for subsequent commands relating to this circuit.

Final return codes:

00H - Success

03H - Invalid command

05H - Connect timed out

09H - No resource available

0BH - Command canceled

11H - Maximum number of sessions

12H - Remote system rejected connection

14H - Called name unknown; this will be returned if either the naming service cannot resolve the name or in the case of a special name; the node is not in the DECnet node database

15H - Invalid name

19H - Name conflict detected

21H - Interface busy

22H - Too many commands

23H - Invalid number in NCB_LANA_NUM

4XH - Network error

80H + nnH - DECnet error

CANCEL

Use this command to cancel a command that was issued previously but not yet completed. Your application must pass the address of the NCB of the command you want to cancel to the NCB_BUFFER field.

Some commands may have progressed to a state that they cannot be canceled. In this case, NETBIOS allows them to complete while the cancel request is pending.

You can CANCEL the following commands:

- ADAPTER STATUS
- CALL
- CHAIN SEND
- FIND NAME
- HANG UP
- LISTEN
- RECEIVE
- RECEIVE ANY
- RECEIVE BROADCAST DATAGRAM
- RECEIVE DATAGRAM
- SEND

The CANCEL command return code indicates the status of the CANCEL command only. To determine the status of the command you want to CANCEL, check the return code for that command.

If you issue a CANCEL for a SEND or CHAIN SEND command, the NETBIOS emulator will abort the link.

Command code:

35H - Synchronous form only.

CANCEL

Input fields:

NCB_LANA_NUM

NCB_BUFFER contains the address of the NCB to cancel.

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

Final return codes:

00H - Command completed

03H - Invalid command

23H - Invalid number in **NCB_LANA_NUM**

24H - Command completed during cancel; this will be the case if the command has progressed beyond the point that it can be canceled.

26H - The requested command is invalid to cancel

4XH - Network error

80H + nnH - DECnet error

CHAIN SEND

Use this command to chain two data buffers together for transmission. NETBIOS concatenates the data in the second buffer to the data in the first buffer, then sends both as a single message.

Your application must specify the length and address of the second buffer in the `NCB_CALLNAME` field. Specify the length in the first two bytes and the address in the next four bytes.

If NETBIOS cannot deliver the message, it aborts the link and returns the "session aborted" status message to your application.

Command code:

17H - Synchronous request.
97H - Asynchronous request.

Input fields:

`NCB_LSN` specifies the logical link on which to send the data.

`NCB_BUFFER` address of first buffer.

`NCB_LENGTH` length of first buffer.

`NCB_CALLNAME` contains the following information:

2 bytes - length of second buffer
4 bytes - address of second buffer

`NCB_POST` address of callback routine.

`NCB_LANA_NUM`

CHAIN SEND

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

Final return values:

- 00H - Success
- 03H - Invalid command
- 08H - Illegal logical link number
- 18H - Session aborted
- 22H - Too many commands outstanding
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

DELETE NAME

Use this command to remove a 16-character name from the local name table.

If there are any logical links for this name, the emulator returns a status indicating active sessions (0FH). NETBIOS flags the entry as “de-registered” but does not remove it from the adapter name table until all sessions associated with this name are closed.

The NETBIOS immediately terminates the following pending commands:

- LISTEN
- RECEIVE ANY
- DATAGRAM RECEIVE
- RECEIVE BROADCAST DATAGRAM

If any of these commands are terminated, NETBIOS returns the NAME DELETED (17H) status message to the application.

Command code:

31H - Synchronous request.

B1H - Asynchronous request.

Input fields:

NCB_NAME the name being deleted.

NCB_POST the address of a callback routine.

NCB_LANA_NUM

DELETE NAME

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

Final return codes:

00H - Success

03H - Invalid command

0FH - Command completed, name has active sessions; the name is flagged as deregistered

15H - Name not found or invalid

21H - Interface busy

22H - All NCBs in use

23H - Invalid number in NCB_LANA_NUM

4XH - Network error

80H +nnH - DECnet error

FIND NAME

Use this command to find the location of a 16-character name on the network. This name is specified in the NCB_CALLNAME field.

NETBIOS sends a name query request on the network. If a remote node has the requested name registered, it responds with information indicating how the name is registered (unique or group).

If no response is received within the system timeout period, the NETBIOS emulator returns a "command timed out" status code. If a response is received, NETBIOS returns the number of nodes that responded, followed by the first or only LAN header from each responding node. This returned data is located at the buffer address specified by the NCB_BUFFER field. The NCB_LENGTH field indicates the number of bytes of data stored.

Command code:

F8H - Synchronous request.

78H - Asynchronous request.

Input fields:

NCB_LANA_NUM

NCB_LENGTH length of data in NCB_BUFFER.

NCB_BUFFER

NCB_POST

NCB_CALLNAME

FIND NAME

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

NCB_RESERVE

NCB_LENGTH

Final return codes:

00H - Success

03H - Invalid command

21H - Interface busy

22H - Too many commands outstanding, all NCBs are being used

23H - Invalid number in NCB_LANA_NUM

4XH - Network error FXH - Unusual condition/error

80H + nnH - DECnet error

HANG UP

Use this command to close a session on the network. NETBIOS will close the circuit you specify in the NCB_LSN field of the NCB.

If this field contains an illegal value, NETBIOS will return an invalid virtual circuit status (0AH) to your application. When NETBIOS receives a HANG UP command from an application, it terminates any pending RECEIVE and RECEIVE ANY commands and returns a "session closed" status message to the application in the NCB_RETCODE field.

Command code:

- 12H - Synchronous request.
- 92H - Asynchronous request.

Input fields:

NCB_LSN specifies the logical link to hang up.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

HANG UP

Final return value:

- 00H - Success
- 03H - Invalid command
- 08H - Invalid logical link
- 0AH - Session closed
- 22H - Too many commands
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

LISTEN

Use this command to listen for a connect request from another application. You can listen for a request from a specific name or from any name. You must specify the name in the `NCB_NAME` field of the NCB or use an asterisk (*) to indicate any name. If you listen for a connect from any name, NETBIOS will return the name of that application, when the connection is made, in the `NCB_CALLNAME` field of the NCB.

You can also specify timeouts for all `RECEIVE` commands issued on the virtual circuit created with the `LISTEN` command. Specify timeout intervals in 500-millisecond units. If you specify zero in these timeout fields, no timeout will occur.

Listening for a specific name has priority over a listen for any name. You can establish a session with either a local or remote name and establish multiple sessions with the same pair of names.

When a `LISTEN` is completed, NETBIOS returns a number from 1 to 254 to the `NCB_LSN` field. NETBIOS uses this number for future references to this connection.

Command code:

- 11H - Synchronous request.
- 91H - Asynchronous request.

Input fields:

`NCB_CALLNAME` either an asterisk if the listen is for anyone, or a specific name.

`NCB_NAME` the local name for which the `LISTEN` is being issued.

`NCB_RTO` a timeout in 500-ms intervals to be applied to receives on any logical links created.

`NCB_STO` not used.

LISTEN

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_LSN number of the logical link, 1 to 254.

NCB_CALLNAME the name that connected, if an asterisk was originally supplied.

NCB_CMD_CPLT

NCB_RETCODE

Final return codes:

00H - Success

03H - Invalid command

0BH - Command canceled

11H - Local name table full

15H - Name was not in local table.

17H - Name deleted, a DELETE NAME command removed the name from the local name table

19H - Name conflict detected

21H - Interface busy

22H - Too many commands, the maximum number specified in a RESET was exceeded

23H - Invalid number in NCB_LANA_NUM

4XH - Network error

80H + nnH - DECnet error

RECEIVE

Use this command to receive data from a specified session. NETBIOS always processes **RECEIVE** commands before **RECEIVE ANY** commands in first-in, first-out order. **RECEIVE** commands may time out, but your application can specify timeout values only during a **CALL** or **LISTEN** command — not on the **RECEIVE**.

If the **RECEIVE** buffer is not large enough for data being sent, NETBIOS returns the “message incomplete” status message to your application in the **NCB_RETCODE** field of the NCB. If this occurs, your application can issue another **RECEIVE** to obtain the rest of the message.

Command code:

- 15H - Synchronous request.
- 95H - Asynchronous request.

Input fields:

NCB_LSN specifies the logical link on which to receive the data.

NCB_BUFFER address receive data buffer.

NCB_LENGTH size of the receive buffer.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_LENGTH the number of bytes returned.

NCB_RETCODE

NCB_CMD_CPLT

RECEIVE

Final return values:

- 00H - Successful
- 03H - Invalid command
- 05H - Command timed out
- 06H - Message incomplete; returned if the buffer not large enough to accommodate the data
- 08H - Illegal logical link number
- 0AH - Session closed
- 0BH - Command canceled
- 18H - Session ended abnormally
- 22H - Too many commands
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

RECEIVE ANY

Use this command to receive data from any active session. You must specify a valid session number in the NCB_NUM field, instead of your name, for this command. This number was returned when you issued the ADD NAME or ADD GROUP NAME command. If the application enters 0xFF in the name field instead of its own name number, then the RECEIVE ANY can be satisfied by any message going to any session on the local system. Note that there is a potential of receiving data meant for other applications if you use 0xFF instead of your session number.

NOTE

Using RECEIVE ANY with 0xFF is not recommended when using active PCSA files servers or file services sessions.

NETBIOS always processes a RECEIVE command before a RECEIVE ANY command, in first-in, first-out order. NETBIOS also receives 0xFF last.

If the RECEIVE buffer is not large enough for data being sent, NETBIOS returns the "message incomplete" status message to your application in the NCB_RETURN_CODE field of the NCB. If this occurs, your application can issue another RECEIVE to obtain the rest of the information.

Command code:

16H - Synchronous request.
96H - Asynchronous request.

Input fields:

NCB_BUFFER address of buffer to receive data.

NCB_LENGTH length of receive buffer.

NCB_NUM session number, or FFH if any circuit.

NCB_POST address of callback routine.

RECEIVE ANY

NCB_LANA_NUM

Output fields:

NCB_LSN the logical link number that the data was received on.

NCB_NUM the adapter name number if FFH was specified as the input value for this field.

NCB_RETCODE

NCB_CMD_CPLT

Final return values:

00H - Success

03H - Invalid command

06H - Message incomplete; returned if buffer was not large enough to contain the data

0AH - Session closed; returned if any commands are queued when the link drops

0BH - Command canceled

13H - Illegal name number

17H - Name deleted; returned if the name is being deleted but has not been removed from the adapter name table

18H - Session ended abnormally

22H - Too many commands outstanding

23H - Invalid number in NCB_LANA_NUM

4XH - Network error

80H + nnH - DECnet error

RECEIVE BROADCAST DATAGRAM

RECEIVE BROADCAST DATAGRAM

Use this command to receive a message from any active NETBIOS session that issued a SEND BROADCAST DATAGRAM. There is no timeout for this command. The command is queued on the receive datagram message queue until data is received.

NOTE

Datagram commands can be implemented only between nodes that are running NETBIOS in a LAN.

Command code:

23H - Synchronous request.

A3H - Asynchronous request.

Input fields:

NCB_BUFFER address of data buffer. Maximum message size is 512 bytes.

NCB_LENGTH size of receive data buffer.

NCB_NUM the local name number issuing the request.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_LENGTH number of bytes returned in buffer.

NCB_CALLNAME is the name which issued the SEND BROADCAST DATAGRAM obtained from the message header.

NCB_RETCODE

NCB_CMD_CPLT

RECEIVE BROADCAST DATAGRAM

Final return value:

- 00H - Success
- 03H - Invalid command
- 06H - Message incomplete
- 0BH - Command canceled
- 13H - Illegal name number
- 17H - Name deleted
- 19H - Name conflict detected
- 21H - Interface busy
- 22H - Too many commands outstanding
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

RECEIVE DATAGRAM

Use this command to receive a datagram message from any name on the LAN.

There is no timeout associated with this command. Your application must have a RECEIVE DATAGRAM command outstanding when the SEND DATAGRAM is issued in order to receive the data.

The RECEIVE DATAGRAM command is queued on a datagram receive queue until data is received.

NOTE

Datagram support is only implemented between nodes running the NETBIOS emulation interface in an Ethernet LAN.

Command code:

21H - Synchronous request.
A1H - Asynchronous request.

Input fields:

NCB_BUFFER address of the buffer into which the received data is placed.

NCB_LENGTH is the size of the receive buffer.

NCB_NUM the local name number for the receive. If this field is FFH then a SEND DATAGRAM message to any local name will satisfy this command.

NCB_POST address of callback routine.

NCB_LANA_NUM

RECEIVE DATAGRAM

Output fields:

NCB_LENGTH the size of the received message.

NCB_CALLNAME the name of the message originator; this field is obtained from the incoming Datagram message header.

NCB_RETCODE

NCB_CMD_CPLT

Final return values:

00H - Success

03H - Invalid command

06H - Message incomplete

0BH - Command canceled

13H - Illegal name number

17H - Name deleted

19H - Name conflict detected

21H - Interface busy

22H - Too many commands outstanding

23H - Invalid number in NCB_LANA_NUM

4XH - Network error

80H + nnH - DECnet error

RESET

Use this command to reset the local status and clear the name and session tables. You should wait until outstanding commands are completed (or you should cancel outstanding commands) before issuing the RESET command.

You can also use this command to delete all names from the local name table.

This command causes NETBIOS to:

- Drop all NETBIOS links
- Restore the NETBIOS emulator to its initial values
- Clear the local name table

It is important to note that the RESET command clears the local name for all applications using NETBIOS emulation. Use caution when using this command.

NOTE

NETBIOS does not support the changing of values for session or command blocks with the RESET command.

RESET

Command code:

32H - Synchronous form only.

Input fields:

NCB_LSN ignored.

NCB_NUM ignored.

NCB_LANA_NUM number of the adapter to reset.

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

Final return codes:

00H - Success

03H - Invalid command

23H - Invalid number in NCB_LANA_NUM

4XH - Network error

80H + nnH - DECnet error

SEND

Use this command to send data on the session you specify by number in the NCB_LSN field on a CALL or LISTEN command. Use the NCB_BUFFER field to specify the data address and the NCB_LENGTH field to specify the data length.

If NETBIOS cannot deliver the message, it aborts the link and returns the "logical link aborted" status message.

Command code:

14H - Synchronous request.

94H - Asynchronous request.

Input fields:

NCB_LSN specifies the logical link on which to send the data.

NCB_BUFFER address of the data to send.

NCB_LENGTH length of data in NCB_BUFFER.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

SEND

Final return codes:

- 00H - Success
- 03H - Invalid command
- 08H - Illegal logical link number
- 0AH - Logical link closed
- 0BH - Command canceled
- 18H - Logical link aborted
- 21H - Interface busy
- 22H - Too many commands
- 23H - Invalid number in NCB_LANA_NUM
- 80H + nnH - DECnet error

SEND BROADCAST DATAGRAM

Use this command to send a message to any active NETBIOS session that has a RECEIVE BROADCAST DATAGRAM command outstanding. NETBIOS processes this command in the same manner as a SEND DATAGRAM with the exception that all local RECEIVE BROADCAST DATAGRAM commands are satisfied by a local SEND BROADCAST DATAGRAM, and the message is always transmitted to the network even if satisfied locally.

NOTE

Datagram support is only implemented between nodes running the NETBIOS emulation interface in an Ethernet LAN.

Command code:

22H - Synchronous command.

A2H - Asynchronous command.

Input fields:

NCB_BUFFER contains the data to be transmitted; maximum of 512 bytes.

NCB_LENGTH the length of the message to be transmitted.

NCB_NUM is the local name number issuing the message.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

SEND BROADCAST DATAGRAM

Final return value:

- 00H - Success
- 03H - Invalid command
- 01H - Illegal buffer length
- 13H - Illegal name number
- 19H - Name conflict detected
- 21H - Interface busy
- 22H - Too many commands outstanding
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

SEND DATAGRAM

Use this command to send a Datagram message to either a unique name or a group name that has a RECEIVE DATAGRAM command outstanding. You must specify a valid number in the NCB_NUM field for your local name, and you must specify the target adapter name or group name in the NCB_CALLNAME field.

NOTE

Datagram support is only implemented between nodes running the NETBIOS emulation interface in an Ethernet LAN.

Command code:

20H - Synchronous request.

A0H - Asynchronous request.

Input fields:

NCB_BUFFER address of data buffer to transmit maximum of 512 bytes of data.

NCB_LENGTH number of bytes to transmit.

NCB_NUM is the local name number that is requesting the SEND DATAGRAM.

NCB_CALLNAME specifies either the 16-byte name or group name to which the message is being sent.

NCB_POST address of the callback routine.

NCB_LANA_NUM

Output fields:

NCB_RETCODE

NCB_CMD_CPLT

SEND DATAGRAM

Final return values:

- 00H - Success
- 03H - Invalid command
- 01H - Illegal buffer length
- 13H - Illegal name number, not found in local adapter name table
- 19H - Name conflict detected
- 22H - Too many commands outstanding
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

SESSION STATUS

Use this command to receive status on all active sessions for a name you specify in the NCB_NAME field. You can also use this command to receive status from all names in the local table, if you specify an asterisk (*) in the first byte of the NCB_NAME field.

NETBIOS returns a 4-byte header plus 36 bytes for each session associated with that name. If the buffer you specify is not large enough to contain the data, NETBIOS will return a "message incomplete" status message to your application in the NCB_RETCODE field of the NCB. Outstanding data is lost.

Command code:

34H - Synchronous request.

B4H - Asynchronous request.

Input fields:

NCB_BUFFER address of data buffer.

NCB_LENGTH size of buffer.

NCB_NAME either a specific local name or * if all names are to be reported on.

NCB_POST address of callback routine.

NCB_LANA_NUM

Output fields:

NCB_LENGTH returns the number of bytes written to the buffer.

NCB_RETCODE

NCB_CMD_CPLT

SESSION STATUS

Final return value:

- 00H - Success
- 03H - Invalid command
- 01H - Illegal buffer length
- 06H - Message incomplete
- 15H - Name not found
- 19H - Name conflict detected
- 21H - Interface busy
- 22H - Too many commands
- 23H - Invalid number in NCB_LANA_NUM
- 4XH - Network error
- 80H + nnH - DECnet error

The data returned has the following format:

SESSION STATUS

Header - 4 bytes

1 byte, name number of the sessions being reported. If a specific name is requested, then this is the name number in the adapter name table; otherwise, 0xFF is returned in this field.

1 byte, the number of sessions with this name, or the total number of sessions if all names are being reported on.

1 byte, number of RECEIVE DATAGRAM and RECEIVE BROADCAST DATAGRAM commands outstanding.

1 byte, number of RECEIVE ANY commands outstanding.

For each session - 36 bytes per session.

1 byte, Session Number.

1 byte, Session State:

01H - LISTEN pending

02H - CALL pending

03H - Session active

04H - HANG UP pending

05H - HANG UP complete

06H - Session aborted

16 bytes, local name.

16 bytes, remote name.

1 byte, number of RECEIVE commands outstanding.

1 byte, number of SEND and CHAIN SEND commands outstanding.

7.8.2 DSCB Commands

The remainder of this chapter describes the DSCB commands.

ADD REMOTE NAME

ADD REMOTE NAME

Use this command to add a remote name definition, with optional access control data, to the remote name table. Access strings that are not being supplied must have their length field set to zero.

Command code:

21H

Input fields:

DSCB_NAME
DSCB_NUM
DSCB_OBJTYPE
DSCB_ACC_LEN
DSCB_ACC
DSCB_PASS_LEN
DSCB_PASS
DSCB_USER_LEN
DSCB_USER

Output fields:

DSCB_ERR

Final return codes:

00H - Function completed successfully
01H - Table full
02H - Duplicate node name
03H - Duplicate node number
09H - Out of resources
0CH - No table entry configured
80H + nnH - DECnet error

ADD SERVER NAME

Use this command to add a DECnet node entry to the remote server table.

Command code:

01H

Input fields:

DSCB_NAME 16-byte node name padded with spaces

DSCB_NUM node number

Output fields:

DSCB_ERR DSCB_INDEX

Final return codes:

- 00H - Success
- 01H - Table full
- 02H - Duplicate node name
- 03H - Duplicate node number
- 09H - Out of resources
- 80H + nnH - DECnet error

DELETE ALL REMOTE NAMES

DELETE ALL REMOTE NAMES

Use this command to remove all entries from the remote adapter name table.

Command code:

26H

Input fields:

None

Output fields:

DSCB_ERR

Final return codes:

00H - Function completed successfully

09H - Out of resources

0CH - No table entries configured

80H + nnH - DECnet error

DELETE ALL SERVER NAMES

DELETE ALL SERVER NAMES

Use this command to delete all server names in the remote server table.

Command code:

07H

Input fields:

None

Output fields:

DSCB_ERR

Final return codes:

00H - Success

09H - Out of resources

80H + nnH - DECnet error

DELETE REMOTE NAME

DELETE REMOTE NAME

Use this command to remove a remote name from the remote name table. You must specify the exact name that you want to remove.

Command code:

22H

Input fields:

DSCB_NAME

Output fields:

DSCB_ERR

Final return codes:

00H - Function completed successfully
04H - No entry with given node name
09H - Out of resources
0CH - No table entry configured
80H + nnH - DECnet error

DELETE SERVER GIVEN NODE NAME

DELETE SERVER GIVEN NODE NAME

Use this command to delete a DECnet node entry from the remote server table. Indicate the entry you want to delete by specifying the DECnet node name in the DSCB_NAME field.

Command code:

03H

Input fields:

DSCB_NAME 16-byte node name padded with spaces

Output fields:

DSCB_ERR

Final return codes:

00H - Success
04H - Node name not found
09H - Out of resources
0AH - Cannot delete own node entry
80H + nnH - DECnet error

DELETE SERVER GIVEN NODE NUMBER

DELETE SERVER GIVEN NODE NUMBER

Use this command to remove a DECnet node entry from the remote server table. Indicate the entry you want to delete by specifying the DECnet node number in the DSCB_NUM field.

Command code:

02H

Input fields:

DSCB_NUM node number

Output fields:

DSCB_ERR

Final return codes:

00H - Success

05H - Node number not found

09H - Out of resources

0AH - Cannot delete own node entry

80H + nnH - DECnet error

READ ALL REMOTE NAMES

Use this command to read all remote names in the remote adapter name table. Specify the length of the buffer for this call in the DSCB_ACC_LEN field.

Before you issue this call, use the READ NUMBER OF REMOTE NAMES command to determine the length of the buffer.

Command code:

25H

Input fields:

DSCB_ACC_LEN the length of the control block

Output fields:

DSCB_ERR

DSCB_INDEX contains the number of entries in the remote adapter name table

The following fields are filled, in consecutive bytes, for each entry:

DSCB_NAME
DSCB_NUM
DSCB_OBJTYPE
DSCB_ACC_LEN
DSCB_ACC
DSCB_PASS_LEN
DSCB_PASS
DSCB_USER_LEN
DSCB_USER

READ ALL REMOTE NAMES

Final return codes:

- 00H - Function completed successfully
- 09H - Out of resources
- 0BH - Buffer too small, cannot fill data into buffer area
- 0CH - No table entries configured
- 80H + nnH - DECnet error

READ NUMBER OF REMOTE NAMES

Use this command to get the number of entries in the remote name table. NETBIOS returns the number of entries to the DSCB_INDEX field. You will need this number to determine the size of the buffer required for the READ ALL REMOTE NAMES command.

The sequence of command actions should be:

1. Read the number of remote names.
2. Set the size of the buffer.
3. Read all of the remote names into the buffer.

Command code:

24H

Input fields:

None

Output fields:

DSCB_ERR

DSCB_INDEX contains number of entries

Final return codes:

00H - Function completed successfully
09H - Out of resources
0CH - No table entries configured
80H + nnH - DECnet error

INSTALLATION CHECK

INSTALLATION CHECK

Use this function call before using any Digital-specific calls to access the NET-BIOS emulator. This function call ensures that the NETBIOS emulation software that supports Digital's extended functions is installed.

To get the version number of the NETBIOS emulator you have installed, use the ADAPTER STATUS function call.

Command code:

00H

Input fields:

DSCB_ERR = 1

Output fields:

DSCB_ERR

DSCB_NAME first double word contains far pointer to path of DECNODE.DAT

Final return codes:

00H - Function completed successfully

09H - Out of resources

80H + nnH - DECnet error

READ REMOTE NAME

Use this command to read an adapter name from the remote adapter name table, by name, and with access control data.

Command code:

23H

Input fields:

DSCB_NAME 16-byte node name padded with spaces

Output fields:

DSCB_ERR
DSCB_NUM
DSCB_OBJTYPE
DSCB_ACC_LEN
DSCB_ACC
DSCB_PASS_LEN
DSCB_PASS
DSCB_USER_LEN
DSCB_USER

Final return codes:

00H - Function completed successfully
05H - No entry with given node number
09H - Out of resources
80H + nnH - DECnet error

READ SERVER BY INDEX

READ SERVER BY INDEX

Use this command to read a DECnet node entry in the remote server table. Indicate the entry you want to read by specifying the index in the DSCB_INDEX field. NET-BIOS returns the corresponding node name and node number to your application.

Command code:

06H

Input fields:

DSCB_INDEX

Output fields:

DSCB_ERR

DSCB_NAME 16-byte node name padded with spaces

DSCB_NUM

Final return codes:

00H - Success

06H - Entry not found

07H - Index out of range

09H - Out of resources

80H + nnH - DECnet error

READ SERVER GIVEN NODE NAME

READ SERVER GIVEN NODE NAME

Use this command to find a DECnet node entry in the remote server table. Indicate the entry you want to read by specifying the node name in the DSCB_NAME field. NETBIOS searches the table and returns the index to the table along with the node number that corresponds with the node name you specified.

Command code:

05H

Input fields:

DSCB_NAME 16-byte node name padded with spaces

Output fields:

DSCB_ERR
DSCB_INDEX
DSCB_NUM

Final return codes:

00H - Success
04H - No entry with given node name
09H - Out of resources
80H + nnH - DECnet error

READ SERVER GIVEN NODE NUMBER

READ SERVER GIVEN NODE NUMBER

Use this command to find a DECnet node entry in the remote server table. Indicate the entry you want to read by specifying the node number in the DSCB_NUM field. NETBIOS returns the index to the table and the node name that corresponds with the node number you specified.

Command code:

04H

Input fields:

DSCB_NUM node number

Output fields:

DSCB_ERR

DSCB_INDEX index into node table

DSCB_NAME 16-byte node name

Final return codes:

00H - Success

05H - No entry with given node number

09H - Out of resources

80H + nnH - DECnet error

Command Terminal (CTERM) Interface

The Command Terminal (CTERM) protocol is a software module that provides a means for terminal emulators and other programs to communicate with a host system in a DECnet™ network.

This chapter describes the functional interface to the CTERM module for use by DECnet-DOS™ client applications. It does not describe the CTERM protocol.

8.1 Introduction

The CTERM module provides wide area network services to client applications connected to a host through Ethernet or asynchronous DDCMP connections. The host system must be a DECnet Phase IV node and have the CTERM DECnet object installed.

Client programs, such as the DECnet-DOS SETHOST terminal emulator, can access CTERM services. These services include:

- Functions to manage terminal sessions to a remote DECnet node
- Support for multiple interactive terminal sessions

The CTERM interface provides local area services to client applications connected to a host through Ethernet connections. An application gets access to CTERM functions by issuing software interrupts through INT 69H. The CTERM software then uses DNP and the data link interface to access the network.

CTERM software does not interfere with other network uses of a workstation. Applications can concurrently use CTERM services and other interfaces. The CTERM interface supports multiple sessions. With CTERM software and a terminal emulator, a workstation can support interactive VMS™ terminal sessions.

Note that SETHOST will load or unload CTERM for you.

8.2 CTERM Services

Client applications such as the SETHOST terminal emulator can access CTERM services such as the following:

- Open session
- Send character
- Receive character
- Close session

8.3 CTERM Command Line

DECnet-DOS provides the CTERM driver as an MS-DOS® terminate and stay resident module, CTERM.EXE. You can invoke the CTERM software by typing CTERM at the DOS prompt or by adding the CTERM command to a batch file. Be sure that you define a path to CTERM.EXE or that CTERM.EXE lies within your current default directory. Once you invoke CTERM.EXE, subsequent invocations will display the version number of the CTERM software installed; a second copy of the software will not be copied.

The CTERM command line should not exceed one line in length. The following command line switch is provided:

- /U — unload CTERM from memory. Use this switch to remove CTERM from memory to free memory, or before installing a newer version of the CTERM driver. This switch will also close all active CTERM connections. You must remove CTERM from memory before you can install another version of the CTERM software. (Note that CTERM is not loadable in EMS.)

Other switches are also available, such as /D (disable), /R (remove), and /T (trace). However, use of the /R switch is not recommended. For more information about these switches, refer to the DEBUG.DOC file included in your DECnet PCSA Client for DOS kit.

8.4 CTERM Sessions

To start or open a session, your application issues a DOS software interrupt, then passes data to the CTERM driver. This data includes the name of the target host node along with other information necessary to start a CTERM session. The CTERM driver, in turn, establishes the session for your application with the DECnet host computer.

The CTERM driver sends messages over a communications channel called a “logical link.” Each active CTERM session uses one DECnet logical link. The CTERM driver supports a maximum of 32 sessions. Because other DECnet-DOS utilities and processes use up logical links, you should make sure to specify enough links to handle your CTERM sessions in addition to other DECnet processes.

Use the DECnet-DOS Network Control Program (NCP) to increase the number of logical links. You can specify any number up to the maximum CTERM sessions of 32 (NCP may allow you to change the maximum number to 64). However, because each link uses memory space, you should specify only the number of links you really need. Also, if you are using DNP in EMS, you cannot use all of the logical links that are available. Certain DNP restrictions apply. Refer to the *DECnet-DOS Network Management Guide* for more information on using NCP to set logical links.

8.4.1 Specifying Nodes to the CTERM Driver

The CTERM driver establishes a terminal session with the remote system specified by your client application. Your application specifies the system to connect to as part of the node specification string it passes to CTERM with the OPEN SESSION function call. The node specification contains the node number or node name and access control information, if necessary. Access control information can include a user name, a password, and account information.

The CTERM driver calls the DECnet function, *dnet_conn*, to validate the node specification. If the node specification is valid, CTERM makes the connection; if not, CTERM returns an error code to your application.

Refer to Chapters 4 and 5 of this manual for more information on using *dnet_conn* and other DECnet function calls.

8.4.2 Issuing Calls to the CTERM Driver

The CTERM driver uses a data structure, the CTERM Session Control Block (SCB), to exchange information between your application and the host node. Before you start a session, your application must first obtain the size of the SCB and then allocate enough memory or buffer space accordingly. Issue the software interrupt 69H to the CTERM driver to determine the SCB size. The CTERM driver returns the size of the SCB to your application. After your application allocates enough memory or buffer space for the driver to use, you can start a session with the CTERM driver.

To start or open a session, your application issues an interrupt (as a CTERM function call) and passes a pointer to the CTERM SCB and a pointer to the node specification. CTERM returns a handle (a 16-bit integer) to your application as a result of this call. Your application must use this handle to refer to the session when sending subsequent function calls.

NOTE

It is the responsibility of the application to pass valid SCBs and handles across the CTERM interface. Failure to pass valid SCBs and handles can produce unpredictable results.

8.4.3 Exchanging Data

After establishing a session, your application can send or receive data, one character at a time, to and from the CTERM driver. The CTERM driver then sends or receives this data to or from a remote host, over the DECnet network. This data is exchanged in the form of CTERM protocol messages. The CTERM driver will read the next message from the host only after your application has finished reading the characters from the current message. DECnet ensures that no messages are lost.

8.4.4 Closing a CTERM Session

Your application must issue a function call to close the CTERM session. When CTERM receives the CLOSE call from your application, it obtains the DECnet socket number that is currently in use for the CTERM connection. CTERM then issues a close request to DECnet to close the connection associated with that socket number.

Important

Be sure to close the CTERM session explicitly: failure to do so can produce unpredictable results. The CTERM software needs to know when your application has finished a session; otherwise it will still try to use the CTERM SCB after your application exits and frees that memory.

8.5 Sending Calls to the CTERM Driver

As previously described, your application accesses the CTERM driver through software interrupt 69H. Your application must allocate memory for the SCB that the CTERM driver will access during a session. The CTERM driver uses the SCB to store data and to control the flow of data between your PC and the host node.

Your application must complete the following process when issuing a call to the CTERM driver:

1. Create an SCB
 - Poll the CTERM driver for the size of the SCB with the GET SCB SIZE function call
 - Allocate memory for an SCB

2. Issue INT 69H.

Your application must provide the address of the CTERM SCB on the OPEN call. The contents of the SCB are known only to the CTERM software; all that the application needs to do is provide an SCB of the correct size. The size is returned from the GET SCB SIZE function call.

8.6 CTERM Functions

The CTERM function calls allow you to communicate with the CTERM driver. Use these calls to initiate a CTERM session. The CTERM services are accessed through INT 69H. The AL register contains the function code for the requested service. Each access requires that 1 is in the AH register. Table 8-1 summarizes the CTERM calls described in this chapter.

Table 8-1: CTERM Functions

Function Code (Hex)	Function Name
00	Installation Check
01	Send Char
02	Read Char
03	Status
04	DECnet Status
05	Open Session
06	Close Session
07 - 09	Reserved
0A	Get SCB Size
0B	Get DECnet Socket
0C - 0E	Reserved
0F	Unload

Table 8-2 lists the CTERM-specific reason codes returned in the AL register.

Table 8-2: CTERM Reason Codes

Reason Code	Description
00H	Normal disconnect
01H	Unknown protocol message from host
02H	Protocol violation by host
03H	Could not process the initiate message
04H	Error receiving message from host
05H	Error sending message to host
06H	Error checking for message from host
07H	Remote system does not support CTERM protocol
08H	Remote system does not support correct protocol version
09H	Did not receive BIND request message from host
0AH	Could not send BIND request message to host
0BH	No more sessions available
0CH	Session does not exist
0DH	Not enough memory to complete operation
0EH	Connection has broken

00H: Installation Check

00H: Installation Check

Use this function call to see if CTERM is installed on your system. The Real-Time Scheduler (SCH) must be installed before you perform this check. You can install the scheduler by typing SCH at the DOS prompt.

On Entry:

AH = 01, module identification

AL = 00, function code

On Return:

If function succeeded:

ES:BX = far address of module entry point

If function failed:

AL = 00, CTERM not installed

AL = 0FFH, CTERM already installed

Each component which handles an interrupt entry for INT 69H will have a mechanism embedded in the interrupt code that identifies the module. This structure will have negative offsets from the module entry point. The format is as follows:

```
DB <major-ver>,<minor-ver>;2-byte version number
DB "CTM";3-byte acronym for module
Interrupt_entry PROC FAR
```

01H: Send Char

Use this function call to send character data to the CTERM driver.

On Entry:

AH = 01, module identification
AL = 01, function call
BL = character to be sent
DX = session handle returned on the OPEN SESSION call

On Return:

If function succeeded:

AH = 0, character sent

If function failed:

AH bit 7 = 1, unable to send char

02H: Read Char

02H: Read Char

Use this function call to receive character data from the CTERM driver.

On Entry:

AH = 01, module identification
AL = 02, function code
DX = session handle returned from open session call

On Return:

If function succeeded:

AL = character received

If function failed:

AH bit 7 = 1, no character read

03H: Status

Use this function call to obtain the status of the CTERM driver during a CTERM session. For example, your application can issue a STATUS call to see whether there is data waiting to be read, or to see if the CTERM session has been aborted.

If the CTERM session aborts, CTERM sets bit 7 in the AH register to indicate this condition. The AL register contains the error code that describes the reason for the failure (for example, AX = 8001). Refer to Table 8-1.

If a DECnet error occurred, CTERM sets bit 6 in addition to bit 7 in the AH register. If this occurs, your application should issue a DECnet status call, function 04H, to obtain more information about the DECnet error (for example, AX = code).

On Entry:

AX = 103H function code

DX = session handle returned from open session call

On Return:

AX = status word

Status is reported in the form of one byte of bit flags followed by one byte of explanation. Bit set (=1) means the condition is true.

AH contains bit flags as described here:

Bit 7	Session has been aborted
Bit 6	DECnet error
Bit 5	Reserved
Bit 4	Reserved
Bit 3	Reserved
Bit 2	Reserved
Bit 1	Reserved
Bit 0	Receive data available

04H: DECnet Status

04H: DECnet Status

Use this function call during a CTERM session to obtain the status of the DECnet network. Call this function only when the CTERM STATUS call returns with the DECnet error bit set.

Use the DECnet library function perror to print a DECnet message that matches the error code.

On Entry:

AH = 01, module identification
AL = 00, function code
DX = session handle returned from open session call

On Return:

AX = DECnet error codes (refer to Appendix G)

05H: Open Session

Use this function call to open a CTERM session. CTERM returns a session handle to your application as a result of this call. Your application must save and use this handle on subsequent calls for this session.

Use the DECnet library function perror to print a DECnet message that matches the error code.

On Entry:

AH	=	01, module identification
AL	=	00, function code
DS:BX	=	long pointer to ASCIZ string nodespec
ES:DX	=	long pointer to the CTERM SCB

On Return:

If function succeeded:

AX	=	an integer greater than 0; handle for subsequent calls to use this session
----	---	--

If function failed:

AX	=	an integer less than or equal to zero; CTERM driver reason code (refer to function 03H: Status for a list of reason codes)
----	---	--

06H: Close Session

06H: Close Session

Use this function call to close a CTERM session. Your application must pass the session handle that CTERM returned on the OPEN SESSION call.

On Entry:

AH = 01, module identification
AL = 06, function code
DL = session handle returned from OPEN SESSION call

On Return:

If function succeeded:

AH = 0

If function failed:

AH = 0CH, session does not exist

0AH: Get SCB Size

Use this function to obtain the size of the CTERM Session Control Block (SCB) data structure. CTERM returns the size to your application as a result of this call.

On Entry:

AH = 01, module identification

AL = 00, function code

On Return:

AX = size of SCB to allocate

0BH: Get DECnet Socket

0BH: Get DECnet Socket

Use this function call to obtain the DECnet socket number associated with this session. CTERM maps the session handle, which was returned to your application on the OPEN call, to the DECnet socket number. You need this socket number if you plan to issue DECnet library function calls.

On Entry:

AH	= 01, module identification
AL	= 0B, function code
AX	= 10BH function code
DX	= session handle returned from OPEN SESSION call

On Return:

If function succeeded:

AX = an integer greater than 0; DECnet socket number for this session

If function failed:

AX = 0, no match

0FH: Deinstall CTERM

Use this function to remove the CTERM driver from memory.

On Entry:

AX = 10FH function code

On Return:

If function succeeded:

AH = 0

If function failed:

AH = 0FFH, could not deinstall CTERM

In order for the Deinstall CTERM to work, CTERM must have been the last installed user of interrupt 69H. The CTERM driver can be removed only if it is the most recent entry in the interrupt chain.

8.7 Sample Terminal Programs

The following are samples of terminal programs that will operate on an IBM PC or industry-compatible personal computer.

8.7.1 Assembler Language Program

TITLE TERMC.ASM - A Sample Terminal program for CTERM in "C".
PAGE 60,132
NAME termc

```
;*****  
;*                                     *  
;* Copyright (c) 1987,1989           *  
;* by DIGITAL Equipment Corporation, Maynard, Mass. *  
;*                                     *  
;* This software is furnished under a license and may be used and copied *  
;* only in accordance with the terms of such license and with the *  
;* inclusion of the above copyright notice. This software or any other *  
;* copies thereof may not be provided or otherwise made available to any *  
;* other person. No title to and ownership of the software is hereby *  
;* transferred. *  
;*                                     *  
;* The information in this software is subject to change without notice *  
;* and should not be construed as a commitment by DIGITAL Equipment *  
;* Corporation. *  
;*                                     *  
;* DIGITAL assumes no responsibility for the use or reliability of its *  
;* software on equipment which is not supplied by DIGITAL. *  
;*                                     *  
;*****  
  
cr        EQU        13        ; Carriage return  
lf        EQU        10        ; Line Feed  
CTERM_INT EQU        69H       ; CTERM shared Interrupt  
  
; Definitions of function codes (ah always = 1)  
  
FC_INSTALL_CHECK       EQU       100H       ; Installation Check  
FC_SEND_DATA        EQU       101H       ; Send Data  
FC_READ_DATA        EQU       102H       ; Read Data  
FC_STATUS        EQU       103H       ; Status  
FC_DECNET_STATUS    EQU       104H       ; DECnet Status  
FC_OPEN_SESSION     EQU       105H       ; Open Session  
FC_CLOSE_SESSION    EQU       106H       ; Close Session  
  
FC_GET_SCB_SIZE     EQU       10AH       ; Get SCB Size  
FC_GET_SOCKET       EQU       10BH       ; Get DECnet Socket  
FC_DEINSTALL       EQU       10FH       ; Deinstall Driver
```

```

;*****
;*
;*      A simple terminal program using the PC CTERM TSR
;*
;*      To build:
;*      MASM TERMC;
;*      LINK TERMC;
;*      EXE2BIN TERMC.EXE TERMC.COM
;*
;*      To invoke, after CTERM is loaded:
;*      TERMC node
;*****

cseg    SEGMENT PUBLIC 'codeseg'

ASSUME  CS:cseg,DS:cseg,ES:cseg,SS:NOTHING

        ORG    100h                ; origin for .COM file

public  start

main    PROC    NEAR

start:

        LEA     BX,cs:lastloc+15    ; Free up memory so that we can allocate
                                      ; a CTERM SCB

        MOV     CL,4
        SHR     BX,CL                ; bytes/16 = paragraphs
        MOV     AX,4A00H            ; Function - free allocated memory
        INT     21h

        CLD                          ; Set to auto-increment.
        MOV     SI,80H              ; Location of command line.
        LODSB                       ; First byte = count in AL

        OR      AL,AL                ; Is it zero?
        JNZ     got_a_name          ; no

        MOV     DX,OFFSET help      ; help message
        MOV     AH,9h               ; Function = write string
        INT     21h                 ; From MS-DOS
        JMP     exit

got_a_name:

        MOV     CL,AL                ; Count in CX
        XOR     CH,CH
        MOV     DI,OFFSET Service   ; Destination in ES:DI
        XOR     CH,CH                ; Zero out CH

copy_loop:

        LODSB                       ; Load first characters.
        CMP     AL,20H               ; Space?
        JE      st_04                ; Ignore spaces

        STOSB                       ; No, save the character.

```

```

st_04:
    LOOP    copy_loop          ; Copy the service name into the scb.
    MOV     AL,CH              ; Terminate the string with a zero byte.
    STOSB                               ; ..

check:
    PUSH    ES
    MOV     AX,FC_INSTALL_CHECK ; Check to see if CTERM is installed.
    INT     cterm_int
    POP     ES
    OR      AL,AL              ; AL.EQ. 0 => not installed
    JNZ     connect           ; Yes, proceed.

    MOV     DX,OFFSET no_cterm_message ; Load pointer to no CTERM message.
    MOV     AH,9               ; Function = write string.
    INT     21h                ; Call MS-DOS
    JMP     error_exit

connect:
    MOV     DX,OFFSET hello_message ; Issue greeting message to user.
    MOV     AH,9h              ; Function = write string.
    INT     21h                ; From MS-DOS

    CALL    cterm_connect      ; Try to connect to that service

main_loop:
    XOR     AX,AX              ; Zero AX and reset flags
    MOV     AH,01H             ; PC Keyboard poll
    INT     16h                ; From the BIOS
    JZ      rxd                ; No character at Kbd, check cterm

;Retrieve character from keyboard buffer
gotkey:
    MOV     AH,00h             ; Function, read character
    INT     16h                ; From the BIOS

;Check it for an Ctrl Backslash = Exit from terminal program
    CMP     AL,1ch             ; Ctrl Backslash Key ?
    JNE     yx_01              ; No

yx_02:
; Close cterm session

    MOV     AX,FC_CLOSE_SESSION ; Function code.
    MOV     DX,WORD PTR handle ; Session handle.
    INT     cterm_int           ; Try to close it.

    MOV     DX,OFFSET session_closed ; Tell user the session is closed
    MOV     AH,9
    INT     21h

    JMP     SHORT exit

```



```

yx_01:
    CMP     AL,08h           ; Backspace?
    JNE     ml_01           ; No, continue.
    MOV     AL,07Fh         ; Yes, map to delete.

ml_01:
    MOV     BX,AX           ; Put the character in BX
    MOV     AX,FC_SEND_DATA ; Function, send data
    MOV     DX,WORD PTR handle ; Use handle given by CTERM
    INT     cterm_int       ; Send the character
    TEST    AH,80h         ; Test for character sent

; See if there is a character received

rx_d:
    MOV     AX,FC_STATUS    ; Check the status
    MOV     DX,WORD PTR handle
    INT     cterm_int
    TEST    AH,80h         ; Session aborted?
    JNZ     circuit_dead
    TEST    AH,1b          ; Character available?
    JZ      main_loop      ; No character, poll keyboard again

; Read the character from the CTERM Driver.

read_data:
    MOV     AX,FC_READ_DATA ; Function code
    MOV     DX,WORD PTR handle ; Use handle to CTERM session
    INT     cterm_int       ; From the CTERM
    TEST    AH,80h         ; Test for character received
    JNZ     main_loop      ; If so, try keyboard!

    OR      AL,AL          ; Null?
    JZ      main_loop      ; If so, don't display

    AND     AL,07Fh        ; Mask out bit 8

    MOV     AH,0Eh         ; PC function, write tty
    XOR     BX,BX          ; Display page 0
    INT     10h            ; PC BIOS write_teletype call
    JMP     main_loop      ; All done, poll keyboard again

exit:
    MOV     AX,4C00h
    INT     21h            ; Normal MS-DOS exit

error_exit:
    MOV     AX,4C01h       ; Error level = 1
    INT     21h            ; Call MS-DOS.

main      ENDP

```

```

cterm_connect  PROC    NEAR

    MOV     AX,FC_GET_SCB_SIZE      ; Find out how big to allocate CTERM SCB
    INT     cterm_int
    MOV     CL,4
    SHR     AX,CL                   ; bytes/16 = paragraphs
    MOV     BX,AX                   ; copy the number of paragraphs into BX
    MOV     AX,4800h                ; Function - Allocate Memory
    INT     21h
    MOV     DX,OFFSET need_memory
    JC      li_02                   ; Not enough memory for this session
    MOV     DX,AX                   ; move the memory block ptr to ES:DX
    MOV     WORD PTR cterm_scb,DX   ; Save it so we can free it later

    MOV     AX,FC_OPEN_SESSION      ; Try to connect
    MOV     BX,OFFSET Service       ; ES:BX is the service to connect to

    INT     cterm_int               ; Invoke CTERM
    OR      AX,AX                   ; Any errors?
    li_go   li_go                   ; AX > zero, no errors.
                                         ; AX .LE. zero, error occurred

li_01:       ; Use this general message for all other failures for now.

    MOV     DX,OFFSET connect_failure ; Send failure message to user.

li_02:

    MOV     AH,9                    ; Function = write string.
    INT     21h                     ; Call MS-DOS
    JMP     SHORT error_exit         ; Exit with error.

li_go:

    MOV     WORD PTR handle,AX      ; Save handle to session

li_exit:

    RET

cterm_connect  ENDP

circuit_dead   PROC    NEAR

    MOV     DX,OFFSET dead_message ; Load offset to circuit dead message.
    MOV     AH,9                    ; Function = write string.
    INT     21h                     ; Call MS-DOS
    JMP     SHORT exit               ; Exit.

circuit_dead   ENDP
PAGE

```

```

handle          DW  0                ; Handle for CTERM session
ctermshb        DW  0

Service         DB  80 DUP (0)       ; Nodespec to connect to

dead_message    DB  cr,lf,'Session disconnected!',cr,lf,'$'
hello_message   DB  cr,lf,'Connecting...press <ctrl-\> to exit',cr,lf,'$'
connect_failure DB  cr,lf,'Connect failed',cr,lf,lf,'$'
no_cterm_message DB  cr,lf,'CTERM not installed',cr,lf,lf,'$'
session_closed  DB  cr,lf,'Session aborted',cr,lf,'$'
need_memory     DB  cr,lf,'Insufficient memory',cr,lf,'$'
help            DB  cr,lf,'Usage: TERMC node',cr,lf,'$'

CSEG            ENDS

lastloc         LABEL                BYTE
                END                  start

```


8.7.2 C Language Sample Program

```
/*
 * TERMC.C      A Sample Terminal program that uses CTERM written in "C".
 */
/*****
 *              Copyright (c) 1987,1988,1989 by              *
 *              Digital Equipment Corporation, Maynard, MA    *
 *              All rights reserved.                          *
 *
 * This software is furnished under a license and may be used and *
 * copied only in accordance with the terms of such license and *
 * with the inclusion of the above copyright notice. This *
 * software or any other copies thereof may not be provided or *
 * otherwise made available to any other person. No title to and *
 * ownership of the software is hereby transferred.            *
 *
 * The information in this software is subject to change without *
 * notice and should not be construed as a commitment by Digital *
 * Equipment Corporation.                                       *
 *
 * Digital assumes no responsibility for the use or reliability *
 * of its software on equipment which is not supplied by Digital. *
 *
 *****/

/*
 * A simple terminal program using the PC CTERM TSR
 *
 * Built with:
 * Microsoft (R) C Optimizing Compiler Version 5.10
 * Microsoft (R) Overlay Linker Version 3.65
 *
 * To build:
 * CL /Zp TERMC.c
 *
 * To invoke, after CTERM is loaded:
 * TERMC node
 */

#include <dos.h>
#include <stdio.h>
#include <signal.h>

/*
 * CTERM driver uses interrupt 69 hex that is shared among several TSR's.
 * AH = 1 is reserved for CTERM; AL is the function code.
 */
#define CTERM 0x69
#define FC_INSTALL_CHECK    0x100    /* Installation Check */
#define FC_SEND_DATA        0x101    /* Send Data */
#define FC_READ_DATA        0x102    /* Read Data */
#define FC_STATUS           0x103    /* Status */
#define FC_DECNET_STATUS    0x104    /* DECnet Status */
#define FC_OPEN_SESSION     0x105    /* Open Session */
```

```

#define FC_CLOSE_SESSION    0x106    /* Close Session          */
#define FC_SCB_SIZE         0x10A    /* Session Control Block Size */
#define FC_SOCKET           0x10B    /* Return DECnet socket     */
#define FC_DEINSTALL        0x10F    /* Deinstall Driver        */

unsigned int size, handle, ch;
char *scb;
int status;
union REGS regs;

main(argc, argv)
int argc;
char *argv[];
{
    /* If no node specified, display help message */
    if (argc < 2)
    {
        printf("\nUsage: TERMC node\n");
        exit(1);
    }

    /* Tell MS-DOS and C to ignore Control-C */
    signal(SIGINT, SIG_IGN);

    /* Check if CTERM is already installed */
    /* Note: this assumes an int69 stub handler installed */
    regs.x.ax = FC_INSTALL_CHECK;

    if (!(int86(CTERM, &regs, &regs) & 1))
    {
        printf("CTERM not installed\n");
        exit(1);
    }

    /* Get the size of the Session Control Block and allocate it */
    regs.x.ax = FC_SCB_SIZE;
    size = int86(CTERM, &regs, &regs);

    if ((scb = (char *)malloc(size)) == NULL)
    {
        printf("\nCould not allocate a SCB of %d bytes\n", size);
        exit(1);
    }

    /* Attempt to connect to the host system */
    printf("Connecting to %s...\n", argv[1]);

    regs.x.ax = FC_OPEN_SESSION;
    regs.x.bx = FP_OFF (argv[1]);
    regs.x.dx = FP_OFF (scb);
    if ((handle = int86(CTERM, &regs, &regs)) < 0)
    {

```

```

printf("\nConnect failed: status = %d\n",handle);
regs.x.ax = FC_CLOSE_SESSION;
regs.x.dx = handle;
int86(CTERM, &regs, &regs);
free(scb);
exit(1);
}
else
{
    /* Wow - we're connected! */
    printf("Logout or Press <Ctrl-\\> to exit\n");
}
while (1)
{
    /* Check if the link is still there */
    regs.x.ax = FC_STATUS;
    regs.x.dx = handle;
    if ((status = int86(CTERM, &regs, &regs)) < 0)
    {
        printf("\nSession disconnected! status = %X\n",status);
        break;
    }
    /* Check if character is available and display it */
    regs.x.ax = FC_READ_DATA;
    regs.x.dx = handle;
    if ((status = int86(CTERM, &regs, &regs)) >= 0)
    {
        if ((status & 0xff) != 0)
            putchar(status & 0xff);
    }
    /* Check if a key has been pressed and process it */
    if (kbhit())
    {
        ch = getch() & 0xff;
        if (ch == 8) /* map backspace (^H) to delete */
            ch = 127;
        if (ch == 0x1C)
        {
            printf("\nSession aborted\n");
            break;
        }
        /* Send the character */
        regs.x.ax = FC_SEND_DATA;
        regs.h.bl = ch;
        regs.x.dx = handle;
        status = int86(CTERM, &regs, &regs);
    }
}

```



```
/* Close the session */  
regs.x.ax = FC_CLOSE_SESSION;  
regs.x.dx = handle;  
int86(CTERM, &regs, &regs);  
free(scb);  
exit(0);      /* exit with no errors */  
}
```

Local Area Transport (LAT) Interface

The Local Area Transport (LAT) module provides network services to terminal emulators and other programs in a Local Area Network (LAN) environment. Client programs can use LAT to communicate with a remote host system over a ThinWire™ or standard Ethernet network. This eliminates the need for serial communications cabling between a personal computer and a host system (such as a VMS™, ULTRIX™, or RSX™ system).

The LAT protocol is a Digital proprietary protocol, which can be licensed. This chapter describes the programming interface to LAT; it does not describe the LAT protocol itself.

9.1 Introduction

The LAT interface provides local area services to client applications connected to a host through Ethernet connections. An application gets access to LAT functions by issuing software interrupts through INT 6AH. The LAT software then uses data link interface to access the network.

LAT software does not interfere with other network uses of a workstation. Applications can concurrently use LAT services and other interfaces. The LAT interface supports multiple sessions on one or more virtual circuits. LAT also supports other related services, such as simple data transfers with appropriate software. With LAT software and a terminal emulator, a workstation can support interactive VMS terminal sessions.

Note that SETHOST will load or unload LAT for you.

9.2 LAT Services

Client applications such as the SETHOST terminal emulator can access LAT services such as the following:

- Session start
- Data exchange
- Flow control

The LAT subsystem provides a service directory facility that supports VAXcluster services. In a VAXcluster, users log on to a service offered by one or more cooperating DECnet nodes instead of to a specific node. Users can log on to the cluster even when a particular node is unavailable or overloaded. Individual nodes in a VAXcluster can also be designated as services to facilitate user logins to that particular node.

The LAT software running on the workstation chooses among the multiple services available with the same name. The software then selects the service with the highest service rating. If more than one node in a VAXcluster offers the service, the system manager can balance the load by using dynamic service ratings. This helps to ensure that the highest service rating is on the least busy node.

9.2.1 LAT Command Line

DECnet-DOS™ provides the LAT software as an MS-DOS® terminate and stay resident module, LAT.EXE. You invoke the LAT software by using the command LAT. You can enter the command at the DOS prompt or add it to a batch file. Be sure that LAT.EXE is in your current default directory or that you define the path to it.

Once you invoke LAT, subsequent invocations will be ignored. If insufficient memory is present, LAT exits with an error code of 8 returned to MS-DOS. No error message is issued to the user.

The LAT command has five command line switches. If you omit a switch or specify a value of -1, the default is used. The command line length cannot exceed one line.

The command line switches are as follows:

- `/D:nn`

This switch increases the default size of the LAT service directory. By default, the LAT service directory has 10 entries, each taking 47 bytes. The *nn* value is an unsigned integer representing a number of additional entries. Each additional entry causes the allocation of an extra 47 bytes of memory, rounded to the nearest 16-byte paragraph. An entry is assumed to be one service offered by one node.

The maximum number of entries is approximately 1000. This reserves approximately 49,072 bytes for the service table. If the overflow call-back is not enabled, the only effect of a service table overflow is that new services are not added to the table. The size of the table is dependent on the version of software you are using, whether or not you are using Expanded Memory (EMS).

- `/G: 1, 2, 3, 32...`

This switch enables selected LAT group codes, thus preventing the processing of unspecified multicast messages. By default, all group codes are enabled. If you use this switch, however, only the specified codes are enabled. This reduces the overhead of servicing multicast service announcement messages and can prevent the service table from filling with unwanted service names. Use this switch only when the number of services routinely exceeds available memory on the workstation.

Each number represents a LAT address group code to enable. Groups are numbered from 0 to 255. For example, to invoke the LAT software and enable group codes 0,2,3 and 54, enter the following:

`LAT /G:0,2,3,54`

- `/N`

This switch allows you to disable LAT multicasts (`/NOMULTICAST`). It is not affected by the `NCP DISABLE MULTICAST` command.

- **/R:n**

This switch sets the number of permitted retransmits for a circuit. By default, eight retransmits are allowed before the circuit is stopped. The *n* value range is 4 to 255.

- **/U**

This switch allows you to unload LAT from memory (if you are using conventional memory). Use of this switch will kill all sessions that are currently running.

NOTE

If you are using Expanded Memory (EMS), you cannot unload LAT.

9.2.2 Service Directory

The LAT software listens to the Ethernet for multicast messages from systems offering LAT services (this includes terminal servers). These messages are transmitted at varying frequencies.

When LAT software receives a LAT multicast message informing it of a service, it adds the service name to the LAT service table, unless the service table is full. In that case, it does not keep the service name, and an error message is returned by a LAT status call. An application can also enable call-back notification for this condition.

A LAT application program reads the service table by using the 6AH software interrupt. Under certain conditions, a LAT server can become unreachable. The LAT software detects this condition when the number of retransmits to the server reaches the maximum specified by the /R switch. If this occurs, the LAT software marks the server as unreachable in the service table and omits its name when reading the table. When the LAT software next receives a multicast LAT service message from the server, the server name becomes readable again.

You can specify that a preferred LAT server be entered into the service table at LAT.EXE start-up time. The LAT software can then use the service name without waiting to receive the server's multicast message. To do this, specify the preferred LAT service names in the DECNODE.DAT file. (You can also use the LATCP convert command that will convert your DECNODE.DAT file to a DECLAT.DAT file, or the LACTP add command to add services to the DECLAT.DAT file.) LAT software attempts to read this file at start-up time. If the number of preferred service names exceeds the size of the start-up service table size, LAT software automatically increases the table size to accommodate the number of entries.

To specify or delete preferred nodes, you can use the DECnet-DOS Network Control Program (NCP) utility. Refer to the *DECnet-DOS Network Management Guide* for more information on NCP. However, using LATCP ADD is the recommended method.

9.2.3 LAT Sessions

A LAT session is conducted between a client application and a LAT server. The LAT software creates one virtual circuit between itself and a particular LAT server. One or more DECnet-DOS applications then communicate with that server over the single virtual circuit.

A LAT session is a unit of data being transmitted or received as a part of the session. Each buffer of data exchanged between an application and LAT server makes up a single session.

LAT software supports up to 4 virtual circuits and 10 sessions.

9.2.4 Starting a Session

To start or open a session, an application passes a pointer to a data structure, the LAT Session Control Block (SCB). The application provides the LAT SCB for use by the LAT software. The SCB stores data and controls data flow between the workstation and the host node.

The SCB contains the name of a desired host service. The first open call to a particular service name creates a virtual circuit, which all subsequent open calls to that service name use. The open call returns an 8-bit integer handle by which subsequent functions refer to the session.

NOTE

The application must pass valid SCBs and handles to the LAT software. Invalid SCBs and handles can produce unpredictable results.

9.2.5 Exchanging Data

After the session is established, the LAT software exchanges data with the application program one character at a time, although it exchanges data with the remote host in multiple-character packets.

To obtain characters, the application uses the READ CHAR and SEND CHAR functions.

9.2.6 Flow Control

Flow control is handled automatically by the protocol. For more information about flow control, refer to the *Local Area Transport (LAT) Network Concepts* guide (order number AA-LD84A-TK).

9.2.7 LAT Call-Back Routines

Application programs can specify an address that the LAT software calls when certain conditions occur. These "call-back" routines are specified in the SCB.

The following considerations apply to all LAT call-back routines.

- Call-back routines are accessed by means of a "far call" and must end with a "far return."
- At the time of the call, interrupts are enabled.
- MS-DOS may be interrupted at call-back time. Do not execute MS-DOS functions from within a call-back routine.
- A call-back on receive routine is executed for each received data buffer (not for each character received).
- A call-back on transmit routine is executed when the transmit buffer is completely emptied by the circuit logic.

9.2.8 Closing the LAT Session

When a session is terminated and its virtual circuit has no other open sessions, the LAT driver shuts down the virtual circuit.

NOTE

Failure to close the LAT session explicitly can produce unpredictable results. The LAT software needs to know when the application session is finished.

9.3 Data Structures

The following data structures communicate information between an application and the LAT software.

9.3.1 The LAT Session Control Block

The LAT SCB structure is set up by the application to control data exchange across the LAT interface. The ES:BX register pair points to the SCB when an Open Session request is issued. After the Open Session request, use the session handle to access LAT software. When a session is open, the SCB cannot be moved. (Since SCB cannot be moved, you cannot use LAT if you are swapping applications in a window environment. The SCB will have to be locked in place.)

This is the structure of the LAT SCB:

SCB STRUC

```
service          DB      18 DUP (0)
node             DB      18 DUP (0)
port             DB      18 DUP (0)
session_stopped  DD      0      ; Address of a call-back routine
table_overflow   DD      0      ; Address of a call-back routine
transmit_notify  DD      0      ; Address of a call-back routine
receive_notify   DD      0      ; Address of a call-back routine
session_status   DW      0
session_state    DW      0
local_credits    DB      0
vcb_ptr          DD      0      ; Pointer to Virtual Circuit Block
                  ; vcb_offset, vcb_segment.
back_session     DW      0
forward_session  DW      0
rem_session_id   DB      0
loc_session_id   DB      0
session_byte_count DB    0
```



```

remote_credits      DB  0
tx_session_data     DB  255 Dup(0) ; Transmit buffer -
                                ; Contains the actual transmit data.
num_sessions        DB  4          ; Number of entries on session_ptr_table.
                                ; Four is the recommended number.
num_occupied        DB  0
next_rx_session     DB  0
cur_buf_session     DB  0
Rx_session_Pntr     DW  0

```

```

session_ptr_table   DW  OFFSET session_1 ; Start of table of 5 session
                                DW  OFFSET session_2 ; buffer offsets.
                                DW  OFFSET session_3
                                DW  OFFSET session_4
                                :           :
                                DW  OFFSET session_n ; Four entries is the recommended
                                                ; table size.

```

```

session_1           DB      259 DUP(0) ; Session buffer 1
session_2           DB      259 DUP(0) ; Session buffer 2
session_3           DB      259 DUP(0) ; Session buffer 3
session_4           DB      259 DUP(0) ; Session buffer 4
:                   :           :
session_n           DB      259 DUP(0) ; Four buffers are recommended.

```

```

SCB                  ENDS

```

The service, which is initialized by the application, is the name of the requested service. The service name is in ASCII format, terminated by a null byte (0). The LAT software converts lowercase to uppercase. The node and port are reserved for future use.

Session_stopped, which is initialized by the application, is the address of a routine to call when the session has been stopped. A value of 0 disables this option.

The LAT call-back routine is entered with the data in Table 9-1.

Table 9-1: LAT Call-Back Routine Data

Function Code	Description
01H	Send Data
02H	Read Data
03H	Get Status
D0H	Open/Close
D1H	Send Break Signal
D2H	Reserved
D3H	Reset LAT Counters
D4H	Copy LAT Counters
D5H	Get Next LAT Service
D6H	LAT Service Table Reset

- **Table_overflow**, which is initialized by the application, is the address of a routine to call when the LAT service table overflows. A value of 0 disables this option. This call-back routine has no data.
- **Transmit_notify**, which is initialized by the application, is the address of a routine to call when the transmit buffer has been completely emptied. A value of zero disables this option. The value can be changed whenever a routine is in place and ready for use.
- **Receive_notify**, which is initialized by the application, is the address of the routine to call when a receive has occurred. It is invoked when the receive buffer is completely full. A value of zero disables this option. The value can be changed whenever a routine is in place and ready for use.
- **Session_status** is the current status of the session. See the section titled *Session Status Word Definition* for a complete description.
- **Session_state** is the protocol engine state of this session. It is used by the LAT software. A value of 0 indicates the halted state.
- **Local_credits**, which you should initialize to zero, is used by the LAT software.

- VCB_pntr is a Long Pointer to this session's Virtual Circuit Block. It is used by the LAT software. The actual format of the pointer is:

VCB_offset DW 0

VCB_segment DW 0

- Back_session is an index to the back SCB on this circuit. It is used by the LAT software.
- Forward_session is an index to the forward SCB on this circuit. It is used by the LAT software.
- Rem_session_id is used by the LAT software.
- Loc_session_id is used by the LAT software.
- Session_byte_count is the number of tx_session_data bytes to be transmitted and used by the LAT software.
- Remote_credits is used by the LAT software.
- Tx_session_data is the transmit buffer that contains actual data to be transmitted to the host.
- Num_sessions, which is initialized by the application, is the number of receive data session buffers.
- Num_occupied must be zero. It is used by the LAT software.
- Next_rx_session must be zero. It is used by the LAT software.
- Cur_buf_session must be zero. It is used by the LAT software.
- Rx_session_pntr, which is initialized by the application, equals (Offset of session_1 entry) + 4. It is used by the LAT software.
- Session_ptr_table is initialized by the application. Each entry points to a session_x field in the following data area. session_ptr_table is used by the LAT software.
- Session_1 through session_n are the receive data buffer areas. They are used by the LAT software. These data buffers must reside completely within the same data segment as the SCB.

You specify the number of receive data buffers. Each receive data buffer uses 259 bytes for the actual buffer and two bytes for session_ptr_table entry. The minimum recommended number of receive data buffers is two. For most configurations, four are adequate.

Refer to Section 9.2.7 for further information on call-back routines. All pointers are in the form of offset – segment.

9.3.2 Session Status Word Definition

This section describes the session_status field of the SCB.

Status is reported in two bytes: one byte of bit flags, followed by one byte of explanation. Bit set (= 1) means that the condition is true.

Status Word – Check for circuit and session state prior to the status call.

Status Word – First Byte – Contains bit flags as follows:

Bit 7–5	Reserved
Bit 4	Host sent a stop session (stop session command)
Bit 3	Circuit failure, reason in second byte
Bit 2–1	Reserved
Bit 0	Transmit buffer busy

Status Word – Second Byte – Contains a reason number code on circuit or session failure as follows:

Code	Explanation
01H	Stop session received.
02H	Stop message received.
03H	Circuit has failed due to excess retransmits.
04H	Illegal session has been received.

- 05H Illegal message has been received.
- 06H User has requested disconnect.
- 07H Stop session received with invalid password.

9.4 LAT Functions

LAT services are accessed through INT 6AH. The AH register contains the function code for the requested service. All calls to the LAT driver require an 0FFH in the DH register.

NOTE

Application software must save and use the session handle for subsequent interface calls. The session handle is returned from the LAT driver on the Open Session call. Using an invalid session handle can cause your workstation to hang.

Table 9-2 lists the available LAT functions. Sections following the table describe these functions.

Table 9-2: LAT Functions

Function Code	Description
01H	Send Char
02H	Read Char
03H	Get Status
D0H	Open/Close
D1H	Send Break Signal
D2H	Reserved
D3H	Reset LAT Counters
D4H	Copy LAT Counters
D5H	Get Next LAT Service
D6H	LAT Service Table Reset

03H: LAT Get Status

Use this function to obtain the status of a LAT session.

On Entry:

AH = 03H

DH = FFH

DL = the session handle returned from the Open Session call

On Return:

AH = status byte (set bits indicate condition)

Bits 7,6 Reserved

Bit 5 Transmit buffer empty

Bit 4 Reserved

Bit 3 Session in start state

Bit 2 Session not active

Bit 1 Unable to queue transmit data

Bit 0 Receive data available

ES:BX = reserved

ES:DX = reserved

D0H: Open Session

D0H: Open Session

Use this function to create buffers and start a LAT session.

On Entry:

AH = D0H

AL = FFH

ES:BX = long pointer to LAT Session Control Block (SCB).

DH = FFH

On Return:

AH = 0H success, or

7-5 Reserved virtual circuit

3 Data buffer specification error

2 No more sessions available

1 No more virtual circuit blocks

0 Service not in table or name error

DL = session handle for subsequent service requests over this session connection

If a virtual circuit to the selected service is not active, a virtual circuit to the node offering the service is created in addition to the requested session.

Application software must save and use the session handle returned from the Open Session call. Failure to use the correct handle can cause unpredictable results.

D0H: Close LAT Session

Use this function to free buffers and close a LAT session.

On Entry:

AH	= D0H
AL	= 000H
DH	= 0FFH
DL	= session handle returned from Open Session call

On Return:

AX	= 0000H, no error
AX	= 0001H, no such active session
AX	= 0002H, session not in running state; try again after a short delay

Before closing the session, the application should confirm that all data has been transmitted. At the close of the session, all receive buffers are freed, even if they still contain data.

02H: Read Char

02H: Read Char

Use this function to read a single byte per call from the LAT driver.

On Entry:

AH = 02H

DH = FFH

DL = session handle returned from Open Session call

On Return:

AL = received character

AH Bit 7 = 1, no character read

01H: Send Char

Use this function to send a single byte per call to the LAT driver.

On Entry:

AL = character to be sent
AH = 01H
DH = FFH
DL = session handle returned from the Open Session call

On Return:

AH = 00H success, or
 Bit 7 = 1, unable to queue character

D5H: Get Next LAT Service Name

D5H: Get Next LAT Service Name

Use this service to read the entries in the LAT service table. To get a list of services, your application must issue successive requests.

The LAT software reports only reachable services. LAT does not report duplicate services or services that are unavailable because they are at an unreachable network node.

On Entry:

AH = D5H

DH = FFH

ES:BX = long pointer to the buffer for the returned service name; the buffer must be at least 17 bytes long

On Return:

AX = 000H success, or

AX = FFFFH, end of table – no service name available

ES:BX = long pointer to the service name terminated by a zero byte

D3H: Reset LAT Counters

Use this function to reset LAT counters to zero.

On Entry:

AH = D3H

DH = FFH

On Return:

LAT counters set to zero

D6H: LAT Service Table Reset

D6H: LAT Service Table Reset

Use this function to reset the LAT service table and LAT counters to zero. A subsequent Get Next LAT Service Name request will return the first entry in the LAT service table.

On Entry:

AH = D6H

DH = 0FFH

On Return:

AX = number of services entered into the service table; this number varies depending on whether or not the service table has been filled

BX = 0, service table not overflowed,

or

= FFFFH, service table has overflowed

D4H: Copy LAT Counters

Use this function to copy LAT counters.

On Entry:

AX = D4H
CX = maximum size of buffer, in bytes
DH = FFH
ES:BX = location of buffer to contain counters

On Return:

AX = 0, counters copied into buffer
AX = FFFFH, buffer size not sufficient

D1H: Send Break Signal

D1H: Send Break Signal

This service is analogous to sending a break signal through a modem.

On Entry:

AH = D1H

DH = 0FFH

DL = session handle returned from Open Session call

On Return:

AX = 0000H, success,

or

AH Bit 7 = 1, unable to send break signal

9.5 Sample Terminal Programs

The following are sample programs that can run on an IBM® personal computer or industry-compatible personal computer.

9.5.1 Assembly Language Terminal Program

TITLE TERML.ASM - A Sample Terminal program for LAT in Assembler.
PAGE 60,132
NAME terml

```
*****
;*
;* Copyright (c) 1985,1989
;* by DIGITAL Equipment Corporation, Maynard, Mass.
;*
;* This software is furnished under a license and may be used and copied
;* only in accordance with the terms of such license and with the
;* inclusion of the above copyright notice. This software or any other
;* copies thereof may not be provided or otherwise made available to any
;* other person. No title to and ownership of the software is hereby
;* transferred.
;*
;* The information in this software is subject to change without notice
;* and should not be construed as a commitment by DIGITAL Equipment
;* Corporation.
;*
;* DIGITAL assumes no responsibility for the use or reliability of its
;* software on equipment which is not supplied by DIGITAL.
;*
*****

cr       EQU       13       ; Carriage return
tab       EQU       9       ; Tab
lf        EQU       10      ; Line Feed
lat_int   EQU       6AH     ; LAT INTerrupt

*****
;*
;*        A simple terminal program using the PC LAT TSR
;*
;*        To build:
;*        MASM TERML;
;*        LINK TERML;
;*        EXE2BIN TERML TERML.COM
;*
;*        To invoke, after LAT is loaded:
;*        TERML service_name
;*
*****

; Dummy segment used to determine if LAT Driver has been installed.

page0       SEGMENT    AT       0
```

```

ORG      lat_int*4                      ; Location of LAT INT in page zero.
lat_entry DD      0
page0    ENDS
cseg     SEGMENT PUBLIC 'codeseg'
ASSUME   CS:cseg,DS:cseg,ES:cseg,SS:NOTHING
ORG      100h                          ; Origin for .COM file
main     PROC NEAR
start:
    MOV    DX,OFFSET hello_message      ; Issue greeting message to user.
    MOV    AH,9                         ; Function = write string.
    INT    21h                          ; From MS-DOS

    CLD                                  ; Set to auto-increment.
    MOV    SI,80H                       ; Location of command line.
    LODSB                                  ; First byte = count in AL

    OR     AL,AL                         ; Is it zero?
    JNZ    got_a_name                   ; no

    MOV    DX,OFFSET help                ; help message
    MOV    AH,9h                         ; Function = write string
    INT    21h                          ; From MS-DOS
    JMP    exit

got_a_name:
    CMP    AL,16                         ; Greater than 16?
    JBE    st_03                         ; Yes, this is an error.
    JMP    service_error

st_03:
    CMP    AL,1                         ; Less than 1?
    JA     st_02                         ; No, we are fine.
    JMP    service_error

st_02:
    MOV    CL,AL                         ; Count in CX
    XOR    CH,CH
    MOV    DI,OFFSET Service             ; Destination in ES:DI
    XOR    CH,CH                         ; Zero out CH

copy_loop:
    LODSB                                  ; Load first characters.
    CMP    AL,20H                       ; Space?
    JE     st_04                         ; Spaces shall be evaporated!
    STOSB                                  ; No, save the character.

```

```

st_04:
    LOOP    copy_loop                ; Copy the service name into the scb.
    MOV     AL,CH                    ; Terminate the string with a zero byte.
    STOSB                                     ; ..

    CALL    check_installation        ; Check to see if LAT is installed.
    JZ      st_01                    ; Yes, proceed.

    MOV     DX,OFFSET no_lat_message ; Load pointer to no lat message.
    MOV     AH,9                     ; Function = write string.
    INT     21h                      ; Call MS-DOS
    JMP     error_exit

st_01:
    CALL    lat_initialization        ; Execute the LAT init call.

main_loop:
    XOR     AX,AX                    ; Zero AX and reset flags
    MOV     AH,01h                   ; Keyboard poll
    INT     16h                      ; From the BIOS
    JZ      rxd                      ; No character at Kbd, check serial port

; Retrieve character from keyboard buffer
    MOV     AH,00h                   ; Function, read character
    INT     16h                      ; From the BIOS

; Check it for an F1 = Exit from terminal program
    CMP     AX,3B00h                 ; F1 Key ?
    JNE     yx_01                    ; No, continue.

; Explicit stop now implemented!

    MOV     AX,0D000h                ; Function, close session.
    MOV     DX,WORD PTR handle       ; Session handle in DX.
    INT     lat_int

JMPexit

yx_01:
    CMP     AX,3C00h                 ; F2 key ?
    JNE     yx_02                    ; No, continue

; Send a break signal
    MOV     AH,0D1h                  ;
    MOV     DX,WORD PTR handle       ;
    INT     lat_int                  ; Send break signal

yx_02:
    CMP     AL,08h                   ; Backspace?
    JNE     ml_01                    ; No, continue.
    MOV     AL,07Fh                  ; Yes, map to delete.

```



```

ml_01:
;Send out the character

MOV     CX,100                ; We will try and xmit 1000 times.
MOV     AH,01h               ;Function, port_write

txd:
MOV     DX,WORD PTR handle    ; Use handle given by LAT
PUSH    AX                   ;Preserve Ax destroyed by LAT INT
INT     lat_int               ;Send the character via the BIOS
TEST    AH,80h               ;Test for character sent

POP     AX                    ;Restore Ax
JZ      rxd                   ;Check for another character
LOOP    txd                   ;Try again if character not sent

;See if there is a character received

rxd:
TEST    session_status,1000b   ; Circuit stopped?
JNZ     circuit_dead

MOV     AH,03h                ; Status
MOV     DX,WORD PTR handle    ; Port
PUSH    ES
INT     lat_int               ; Get status
POP     ES

TEST    AH,01h                ; Character available?
JZ      main_loop             ; No character, poll keyboard again

; Read the character from the LAT buffer.

MOV     DX,WORD PTR handle    ;Use handle to LAT session
MOV     AH,02h                ;Function, port_read
INT     lat_int               ;From the BIOS
TEST    AH,80h                ;Test for character received
JNZ     main_loop             ; If so, try keyboard!

OR      AL,AL                 ; Null?
JZ      main_loop             ; If so, don't display

; CMP     AL,09h               ; Tab?

AND     AL,07Fh               ;Mask out bit 8
MOV     AH,0Eh                ;function, write TTY
XOR     BX,BX                 ;Display page = 0
INT     10h                   ;BIOS write_teletype call
JMP     main_loop             ;All done, poll keyboard again

```

```

exit:
    MOV     AX,4C00h
    INT     21h                      ; Normal MS-DOS exit

service_error:
    MOV     DX,OFFSET bad_service_mess ; Bad service message.
    MOV     AH,9                      ; Function = write string.
    INT     21h                      ; Call MS-DOS.

error_exit:
    MOV     AX,4C01h                  ; Error level = 1
    INT     21h                      ; Call MS-DOS.

main      ENDP

lat_initialization      PROC      NEAR
    MOV     DX,OFF00h                ; This INT is for LAT
    MOV     BX,OFFSET scb            ; ES:BX points to lccb
    MOV     AX,0D0FFh                ; Extended function
    INT     lat_int                  ; Invoke LAT
    OR      AH,AH                    ; Any errors?
    JZ      li_go                    ; AH = zero, no errors.

    TEST    AH,1                     ; Service not in directory?
    JZ      li_01                    ; No.

    MOV     DX,OFFSET no_service_message ; Issue not in directory mes-
    sage.
    MOV     AH,9                      ; Function = write string.
    INT     21h                      ; Call MS-DOS

li_01:: Use this general message for all other failures for now.

    MOV     DX,OFFSET init_failure    ; Send failure message to user.
    MOV     AH,9                      ; Function = write string.
    INT     21h                      ; Call MS-DOS
    JMP     SHORT error_exit          ; Exit with error.

li_go:
    MOV     WORD PTR handle,DX        ; Save handle to session

li_exit:
    RET

lat_initialization      ENDP

circuit_dead      PROC      NEAR
    MOV     DX,OFFSET dead_message    ; Load offset to circuit dead message.
    MOV     AH,9                      ; Function = write string.
    INT     21h                      ; Call MS-DOS
    JMP     SHORT exit                ; Exit.

circuit_dead      ENDP

PAGE

```

```

;*****
;*
;*      PROCEDURE check_installation      *
;*
;*      Entry: Nothing                    *
;*      Exit:  Z flag set = LAT installed. *
;*
;*****
check_installation      PROC      NEAR

    PUSH    AX                ; Preserve registers.
    PUSH    CX                ; .
    PUSH    SI                ; .
    PUSH    DI                ; .
    PUSH    ES                ; ..

    CLD                      ; Set the direction flag to forward.
    XOR     AX,AX             ; Set ES to page zero.
    MOV     ES,AX             ; ..

ASSUME  ES:page0

    LES     DI,DWORD PTR lat_entry      ; ES:DI => lat_int entry

ASSUME  ES:NOTHING

    MOV     CX,3              ; Compare 3 bytes
    SUB     DI,3              ; Starting at entry -3
    MOV     SI,OFFSET lat_string      ; Local string for compare.

    REPZ    CMPSB             ; Compare it!

    POP     ES                ; Restore registers.
    POP     DI                ; .
    POP     SI                ; .
    POP     CX                ; .
    POP     AX                ; ..

    RET

lat_string      DB  'LAT'

check_installation      ENDP

PAGE

dead_message    DB  cr,lf,'Session disconnected!',cr,lf,'$'
hello_message   DB  cr,lf,'Connecting...press <ctrl-\> to exit',cr,lf,'$'
init_failure    DB  cr,lf,'Connect failed',cr,lf,lf,'$'
no_lat_message  DB  cr,lf,'LAT not installed',cr,lf,lf,'$'
no_service_message  DB  cr,lf,'Unknown service',cr,lf,lf,'$'
bad_service_mess  DB  cr,lf,'Illegal Service Name',cr,lf,lf,'$'
help            DB  cr,lf,'Usage: TERML service',cr,lf,'$'

handle      DW      0          ; Handle for LAT session

```



```

;
; SCB = Session Control Block. Structure used by client application
;       to arrange for data exchange.
;

scb          LABEL          WORD

Service      DB              18 DUP (0) ; Requested service.
Node         DB              18 DUP (0) ; Reserved for future use.
Port         DB              18 DUP (0) ; Reserved for future use.

; *****
; The following four call back addresses must be initialized to 0
; if call backs are not desired for each condition.

Session_Stopped      DD      0 ; Session stopped notification routine.
Table_overflow        DD      0 ; Service table overflow notification routine.
Transmit_notify       DD      0 ; Routine to call when slot is transmitted.
Receive_notify        DD      0 ; Routine to call when a slot is received.

session_status        DW      0 ; Status word
slot_state            DW      0 ; Used by LAT Driver - initialize to 0
local_credits         DB      0 ; Used by LAT Driver - initialize to 0

vcb_offset           DW      0 ; Used by LAT Driver. Pointer to LAT
vcb_segment           DW      0 ; Driver's internal circuit block.

back_slot             DW      0 ; Used by LAT Driver - initialize to 0
forward_slot          DW      0 ; Used by LAT Driver - initialize to 0

; Transmit slot buffer - Contains actual transmit slot.

rem_slot_id           DB      0 ; Used by LAT Driver - initialize to 0
loc_slot_id           DB      0 ; Used by LAT Driver - initialize to 0
slot_byte_count       DB      0 ; Used by LAT Driver - initialize to 0
remote_credits        DB      0 ; Used by LAT Driver - initialize to 0

tx_slot_data          Db      255 Dup(0) ; Transmit slot data buffer.
;
; Transmit data area
;
; >>>>>>>> The following variable is initialized by the client application!!
;
Num_slots             DB      4 ; Number of receive data slot buffers
                        ; in this structure. Value of 4 is suggested.

Num_occupied          DB      0 ; Number of occupied slots.
Next_rx_slot          DB      0 ; Index - Next slot to be used for receive slot.
Cur_buf_slot         DB      0 ; Index - Current slot sending characters to client.
;
;>>> The following variable must be initialized by the client application!
; Rx_Slot_Pntr        DW      OFFSET Slot_1+4 ; Offset of the first character
                        ; to be taken by client.
;
;>>> The following table of pointers must be initialized by the client
; application!

```

slot_Ptr_table	LABEL	WORD
	DW	OFFSET slot_1
	DW	OFFSET slot_2
	DW	OFFSET slot_3
	DW	OFFSET slot_4

;>>>>> The following data definitions are the actual receive data buffers.

slot_1	DB	259 DUP(0)
slot_2	DB	259 DUP(0)
slot_3	DB	259 DUP(0)
slot_4	DB	259 DUP(0)

CSEG ENDS

END start

9.5.2 C Language Terminal Program

```
/*
 *
 *      Copyright (c) 1989 by
 *      Digital Equipment Corporation, Maynard, MA
 *      All rights reserved.
 *
 * This software is furnished under a license and may be used and
 * copied only in accordance with the terms of such license and
 * with the inclusion of the above copyright notice. This
 * software or any other copies thereof may not be provided or
 * otherwise made available to any other person. No title to and
 * ownership of the software is hereby transferred.
 *
 * The information in this software is subject to change without
 * notice and should not be construed as a commitment by Digital
 * Equipment Corporation.
 *
 * Digital assumes no responsibility for the use or reliability
 * of its software on equipment which is not supplied by Digital.
 *
 *****/
/*
 * A simple terminal program using the LAT TSR
 *
 * Built with:
 *      Microsoft (R) C Optimizing Compiler Version 5.10
 *      Microsoft (R) Overlay Linker Version 3.65
 *
 * To build, using packed structures:
 *      CL /Zp terml.c
 *
 * To invoke, after LAT is loaded:
 *      TERML servicename
 */

#include <dos.h>
#include <stdio.h>
#include <signal.h>
#include <dos.h>
#include <process.h>
#include <memory.h>
#include <string.h>

#define byte      unsigned char
#define word      unsigned int
#define dword     unsigned long

#define LAT_INT   0x6A      /* LAT software interrupt */
#define DOS_INT   0x21      /* DOS software interrupt */

#define TRUE      1
#define CONTROL_SLASH 0x1C  /* key used to exit TERML */
#define LAT_FLAG  0xFF00    /* identifies call as a LAT call */
```



```

#define TRANSMIT      0x01      /* transmit a character */
#define RECEIVE      0x02      /* receive a character */
#define START_SESSION 0xD0FF    /* start a LAT session */
#define CLOSE_SESSION 0xD000    /* close a LAT session */
#define KBD_INPUT    0x08      /* DOS function number */
#define GET_VECTOR    0x35      /* DOS function number */

#define SLOT_LEN      259      /* slot length */
#define BUFFER_LEN     255     /* transmit buffer length */
#define SRV_LEN        18      /* constant used in SCB */
#define NUM_SLOTS      4       /* number of slots */

byte slot_1[SLOT_LEN] = {0}; /* session slots */
byte slot_2[SLOT_LEN] = {0};
byte slot_3[SLOT_LEN] = {0};
byte slot_4[SLOT_LEN] = {0};

/*
 * SCB = Slot Control Block used for each session
 */

struct scb_tag {
    byte service[SRV_LEN]; /* application must set */
    byte node[SRV_LEN];
    byte port[SRV_LEN];
    dword session_stopped; /* set to zero or address */
    dword table_overflow; /* set to zero or address */
    dword transmit_notify; /* set to zero or address */
    dword receive_notify; /* set to zero or address */
    word session_status; /* session status */
    word slot_state;
    byte local_credits;
    word vcb_offset;
    word vcb_segment;
    word back_slot;
    word forward_slot;
    byte rem_slot_id;
    byte loc_slot_id;
    byte slot_byte_count;
    byte remote_credits;
    byte tx_slot_data[BUFFER_LEN]; /* transmit buffer */
    byte num_slots; /* application must set */
    byte num_occupied; /* application must set to 0 */
    byte next_rx_slot; /* application must set to 0 */
    byte cur_buf_slot; /* application must set to 0 */
    byte *rx_slot_ptr; /* application must set */
    byte *slot_ptr_table[4]; /* application must set */
} scb;

word handle; /* 0xFF00 + session handle */

```

```

/*****
*
*main
*
*****/
main(argc,argv)
int argc;
char *argv[];
{
union REGS inregs, outregs;

    if (argc < 2) {
        printf("\nUsage: TERML service\n");
        exit(1);
    }
    else if (!lat_installed()){
        printf("\nLAT is not installed\n");
        exit(1);
    }

/*
* initialize scb
*/

    memset((char *) &scb, '\0', sizeof(struct scb_tag));
    strcpy(scb.service, argv[1]);          /* set service
    */
    scb.num_slots = NUM_SLOTS;             /* set number of slots
    */
    scb.rx_slot_ptr = slot_1 + 4;          /* set to offset of slot_1 +
    4*/
    scb.slot_ptr_table[0] = slot_1;        /* point to slots
    */
    scb.slot_ptr_table[1] = slot_2;
    scb.slot_ptr_table[2] = slot_3;
    scb.slot_ptr_table[3] = slot_4;

    signal(SIGINT,SIG_IGN);                /* trap <CTRL/C> */
    start_session();                       /* start LAT session */
    while (TRUE){
        display_rcv_char();                /* try to read a character */
        if (kbhit()){                      /* user press a key ?          */
            inregs.h.ah = KBD_INPUT;
            int86(DOS_INT, &inregs, &outregs);
            if (outregs.h.al == CONTROL_SLASH){
                stop_session();
                exit(0);
            }
            else
                send_char(outregs.h.al);
        }
    }
}

```

```

    }
}
exit(0);                                /* exit with no errors */
}

/*****
 *   start_session
 *
 *   Starts a LAT session.
 *****/

start_session(){
union REGS inregs, outregs;
struct SREGS segregs;

    printf("Connecting to %s...\n",strupr(scb.service));

    segread(&segregs);
    inregs.x.dx = LAT_FLAG;
    inregs.x.ax = START_SESSION;
    segregs.es = segregs.ds;
    inregs.x.bx = (word) &scb;          /* ES:BX => SCB */
    int86x(LAT_INT, &inregs, &outregs, &segregs);

    if (outregs.h.ah != 0){
        printf("\nConnect failed. Status = %X hex\n", outregs.h.ah);
        exit(1);
    }
    else {
        handle = outregs.x.dx;
        printf("Logout or Press <Ctrl-\\> to exit\n");
    }
}

/*****
 *   stop_session
 *
 *   Stop the LAT session.
 *****/

stop_session(){
union REGS inregs, outregs;

    inregs.x.ax = CLOSE_SESSION;
    inregs.x.dx = handle;
    int86(LAT_INT, &inregs, &outregs);
    if (outregs.x.ax != 0)
        printf("\nError closing session. Status = %X hex\n", outregs.x.a
x);
}

```



```

/*****
*      display_rcv_char
*
*      Check if a character has been received.  If so display it.
*****/
display_rcv_char(){
union REGS inregs, outregs;

/*
* read a character if its available, and display it
*/

    inregs.h.ah = RECEIVE;
    inregs.x.dx = handle;
    int86(LAT_INT, &inregs, &outregs);
    if (outregs.h.ah < 0x80)          /* error return in AH      */
        putchar(outregs.h.al);      /* character returned in AL */

/*
*      check if session stopped for some reason
*/

    if (scb.session_status & 0x8){
        printf("\nSession disconnected. Status = %X hex\n",
            scb.session_status);
        exit(1);
    }
}

/*****
*      send_char
*
*      Send a character entered at the keyboard
*****/
send_char(kbd_char)
byte kbd_char;
{
union REGS inregs, outregs;

    inregs.h.ah = TRANSMIT;
    inregs.h.al = kbd_char;
    inregs.x.dx = handle;
    int86(LAT_INT, &inregs, &outregs);
}

```

```

/*****
*      lat_installed
*
*      Returns TRUE if LAT is installed. It checks if LAT is installed by
*      getting the vectors address, and then testing for the 'LAT' TLA which
*      should precede it.
*****/
lat_installed(){
byte far *p;
union REGS inregs, outregs;
struct SREGS segregs;
dword es_seg;

    inregs.h.ah = GET_VECTOR;
    inregs.h.al = LAT_INT;
    int86(DOS_INT, &inregs, &outregs);
    segread(&segregs);
    es_seg = (segregs.es << 16L);
    p = (char far *) (es_seg + outregs.x.bx);
    p -= 3;
    if (p[0] != 'L' || p[1] != 'A' || p[2] != 'T')
        return(0);
    else
        return(1);
}

```

Part III

111 7167

A

Socket Definitions

The following definitions are related to socket types, option flags, and other related socket definitions. The symbols that appear in this appendix are defined in the DECnet header files.

A.1 Communications Domain

DECnet-DOS supports the following communications domain:

Decimal Value	Domain	Description
1	AF_DECnet	Enables multiple computer systems to participate in communications and resource sharing within a DECnet network.

The symbol is defined in <socket.h> header file.

A.2 DECnet Layers

The following DECnet layers are supported by DECnet-DOS:

Hexadecimal/ Decimal Value	Layer	Description
0xffff	SOL_SOCKET	Specifies the socket session interface layer.
1	DNPROTO_NSP	Specifies the DECnet layer. How a connection request is accepted/rejected, optional access control and/or user data, or link state can be specified. (See Appendix B for a list of defined data structures.)

These symbols are defined in <socket.h> header file.

A.3 DECnet Objects

Certain DECnet object numbers are used as arguments to the *dnet_conn* call. The following are ASCII strings:

Object	ASCII String
DNOBJ_FAL	#17 (File Access Listener)
DNOBJ_NICE	#19 (Network Information and Control Exchange)
DNOBJ_TERM	#23 (Network command terminal handler - host side)
DNOBJ_MIRROR	#25 (Loopback mirror - MIR)
DNOBJ_EVR	#26 (Event receiver - EVR)
DNOBJ_MAIL11	#27 (Personal message utility)
DNOBJ_PHONE	#29 (Phone utility)
DNOBJ_CTERM	#42 (Command terminal operations)
DNOBJ_DTR	#63 (DECnet test receiver tool -DTR)

The following are decimal numbers:

Decimal Value	Object	Process Type
17	DNOBJECT_FAL	File Access Listener
19	DNOBJECT_NICE	Network Information and Control Exchange
23	DNOBJECT_DTERM	Network command terminal handler - host side
25	DNOBJECT_MIRROR	Loopback mirror (MIR)
26	DNOBJECT_EVR	Event receiver (EVR)
27	DNOBJECT_MAIL11	Personal message utility
29	DNOBJECT_PHONE	Phone utility
42	DNOBJECT_CTERM	Command terminal operations
63	DNOBJECT_DTR	DECnet test receiver tool (DTR)

These symbols are defined in <dn.h> header file.

A.4 DECnet Options

At the DECnet layer (DNPROTO_NSP), socket options can define how a connection request is accepted/rejected, specify optional user data and/or access control information, or obtain current link state information. The following options can be used to specify or retrieve data with the *setsockopt* and *getsockopt* function calls:

Decimal Value	Option	Description
1	DSO_CONDATA	Allows up to 16 bytes of optional user data to be set by the <i>setsockopt</i> call. The optional data is passed in the <i>optdata_dn</i> data structure. The user task reads the data by issuing the <i>getsockopt</i> call with the connect option. The call returns a connect status.
2	DSO_DISDATA	Allows up to 16 bytes of optional user data to be set by the <i>setsockopt</i> call. The optional data is passed in the <i>optdata_dn</i> data structure. The user task reads the data by issuing the <i>getsockopt</i> call with the disconnect option. The call returns a disconnect status.

Decimal Value	Option	Description
3	DSO_CONACCESS	Allows access control information to be set by the <i>setsockopt</i> call. The access control information is passed in the <i>accessdata_dn</i> data structure. The user task reads the data by issuing the <i>getsockopt</i> call. The information is processed once the task issues the <i>accept</i> call.
4	DSO_ACCEPTMODE	Defines the way in which a user task accepts a pending <i>accept</i> call. A socket must issue a <i>bind</i> call before this option is valid. The acceptance mode can be specified as follows:
0	ACC_IMMED	Specifies the default condition. When immediate mode is in effect, control is immediately returned to the server task following an <i>accept</i> call with the connection request accepted. The access control information and/or optional user data is ignored by the server task. The <i>accept</i> call is immediately completed.
1	ACC_DEFER	Allows the server task to complete the <i>accept</i> call without fully completing the connection to the client task. The server task can examine the source address, access control, and/or optional user data before accepting or rejecting the pending connection.
2	ACC_REUSE	Enables your application to reuse the listening socket to receive incoming connections. To receive the connect, your application must issue an <i>accept</i> call. Your application must supply the socket number, the same value as for the listening socket, or specify zero. Specifying a zero causes DNP to return the correct socket number.
5	DSO_CONACCEPT	Allows the server task to accept the pending connection on the socket previously set to the deferred accept mode (ACC_DEFER). Any optional user data previously set by DSO_CONDATA will also be sent.
6	DSO_CONREJECT	Allows the server task to reject the pending connection on the socket previously set to the deferred accept mode (ACC_DEFER). Any optional user data previously set by DSO_DISDATA will also be sent.
7	DSO_LINKINFO	Allows the user task to retrieve the state of the logical link connection. There are four supported

Decimal Value	Option	Description
		link states. (See Section A.6.) The link state is returned in the <i>linkinfo_dn</i> data structure. It is retrieved with the <i>getsockopt</i> call.
10	DSO_LINKHOLD	Allows the user to specify a retry period, in seconds, in addition to normal NSP retries.
10	DSO_MAX	Specifies the allowable number of defined socket options.
11	DSO_FLOWCTRL	Allows NSP to use flow control techniques. Two settings are available: NoFlow or Segment Count.. These values apply: 0 = NoFlow (XON/XOFF or SEND/DONTSEND) 1 = Segment Count (default)

These symbols are defined in <dn.h> header file.

A.5 Flag Options

The following bits can be set in the *io_flags* field which is included in the IOCB and/or CIOCB:

Hexadecimal Value	Message	Description
0x0001	MSG_OOB	Process out-of-band messages with the <i>send</i> and <i>recv</i> calls.
0x0002	MSG_PEEK	Read the next pending message without removing the message from the receive queue.
0x0008	MSG_ASYNC	Process the asynchronous I/O form of DECnet function calls.
0x0010	MSG_CALLBACK	Specifies that the network will issue a callback routine when the function call completes.
0x0020	MSG_NEOM	For sending or receiving a single message in multiple operations. For sending, this flag indicates that this data is not the end of the message. For receiving, this flag indicates that the remaining message data should be saved for a later call.
0x0040	MSG_NBOM	For sending or receiving a single message in multiple operations. This flag indicates that this data is not the beginning of the message. If MSG_NEOM is also set, MSG_NBOM indicates the middle portion of the message.
0x0080	MSG_NIOCB	Flag each call to indicate new NIOCB. Must be set to one.
0x0100	MSG_USRBUF	Flag to indicate use of user-supplied buffer for sending or receiving large data buffers (up to 64KB).
0x0200	MSG_USRWAIT	This flag tells DNP not to wait for completion of an operation on a blocking socket. The user must check the <i>io_status</i> field for completion status.

A.6 Logical Link States

The following logical link states are supported by DECnet-DOS.

Decimal Value	State	Description
0	LL_INACTIVE	The logical link is inactive.
1	LL_CONNECTING	The logical link is connecting.
2	LL_RUNNING	The logical link is running.
3	LL_DISCONNECTING	The logical link is disconnecting.

The symbols are defined in <dn.h> header file.

A.7 Maximum Number of Incoming Connection Requests

The maximum number of incoming connection requests is specified as follows:

Hexadecimal Value	Message	Description
0x5	SO_MAXCONN	Defines the maximum number of incoming connection requests which are allowed on the specified socket.

The symbol is defined in <socket.h> header file.

A.8 Socket Interface Options

At the socket level (SOL_SOCKET), the following options exist:

Hexadecimal Value	Flag	Description
0x04	SO_REUSEADDR	Allows the reuse of a bound socket name. This option must be used only for outgoing connection requests.
0x08	SO_KEEPAIVE	If this option is set on a socket, any links and sockets associated with this socket remain active, despite any attempts to abort, detach, and/or disconnect them. The effects of ABORT, DETACH, and DISCONNECT functions are only realized after SO_KEEPAIVE is turned off.
0x80	SO_LINGER	Controls the actions taken when unsent messages are queued on a socket and the <i>sclose</i> (or the <i>disconnect</i>) call is issued. If SO_LINGER is set, the connection is maintained until the outstanding messages have been sent.
~SO_LINGER	SO_DONTLINGER	Controls the actions of unsent messages. If SO_DONTLINGER is set, and the <i>sclose</i> (or the <i>disconnect</i>) call is issued, any outstanding messages queued to be sent will be lost. The connection is then terminated.
0100	SO_RCVUSRBUF	Specifies that Select functions will complete a read-ready condition upon the first receipt of data on the requested socket, instead of waiting for the entire message to arrive. Can be used in conjunction with Receive and MSG_USRBUF flag to receive large messages without requiring the network to buffer the entire message.

The symbols are defined in <socket.h> header file.

A.9 Socket Types

DECnet-DOS supports the following socket types:

Decimal Value	Type	Description
1	SOCK_STREAM	Stream sockets cause bytes to accumulate until internal DECnet buffers are full. The receiving task does not know how many bytes were sent in each write operation.
5	SOCK_SEQPACKET	Sequenced sockets cause bytes to be sent immediately. The receiving task receives those bytes in one "record".

The symbols are defined in <socket.h> header file.

A.10 Defined Software Modules

The following software modules are supported by DECnet-DOS. They have defined three letter acronym (TLA) strings.

Module Name	TLA String	Description
DNMOD_SES	SES	MS-NET to DECnet Session Interface
DNMOD_LAT	LAT	LAT Driver
DNMOD_PDV	PDV	Port Driver
DNMOD_SCH	SCH	Real-Time Scheduler
DNMOD_DLL	DLL	Data Link Layer
DNMOD_DNP	DNP	DECnet Network Process

The following interrupt vectors have been defined for these DECnet-DOS software modules:

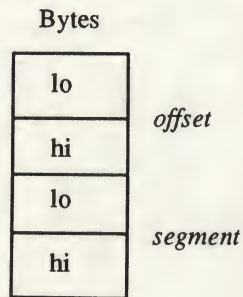
Vector Number (Hex)	Symbol	Description
0x2A	DNMODULE_SES	MS-NET to DECnet Session Interface
0x6A	DNMODULE_LAT	LAT Driver
0x6B	DNMODULE_PDV	Port Driver
0x6C	DNMODULE_SCH	Real-Time Scheduler
0x6D	DNMODULE_DLL	Data Link Layer
0x6E	DNMODULE_DNP	DECnet Network Process

The symbols are defined in <dn.h> header file.

Defined Data Structures and Data Members

The following data structures can be used with specific socket interface and assembly language network driver interface calls. Guidelines for specifying a data structure are detailed with the appropriate function call. The symbols that appear in this appendix are defined in the DECnet header files.

If data type *exptr* is used as an address (or a far pointer), it takes the following format. It is defined in <types.h> header file.



B.1 Access Control Information Data Structure

The *accessdata_dn* data structure contains the following data members:

Type	Data Size	Member	Description
unsigned short	2 bytes	<i>acc_accl</i>	Defines the length of the account string.
unsigned char	40-byte array	<i>acc_acc</i>	Specifies the account string.
unsigned short	2 bytes	<i>acc_passl</i>	Defines the length of the password string.
unsigned char	40-byte array	<i>acc_pass</i>	Specifies the password string.
unsigned short	2 bytes	<i>acc_userl</i>	Defines the length of the user ID string.
unsigned char	40-byte array	<i>acc_user</i>	Specifies the user ID string.

The symbols are defined in the <dn.h> header file.

B.2 Attach Data Structure

The *attach_dn* data structure contains the following data members:

Type	Size	Data Member	Description
int	2 bytes	<i>att_socket</i>	Specifies the number of the socket. If nonzero, the other data structure members are ignored.
unsigned short	2 bytes	<i>att_domain</i>	Specifies the communications domain for the socket as AF_DECnet.
unsigned short	2 bytes	<i>att_type</i>	Specifies the socket type for the socket. For example, SOCK_STREAM. (See Appendix A for a list of defined socket types.)
unsigned short	2 bytes	<i>att_protocol</i>	Specifies the protocol for the socket. For example, DNPROTO_NSP. (See Appendix A for a list of defined protocol interfaces.)
unsigned short	2 bytes	<i>att_srp</i>	Specifies the socket recovery period.
unsigned short	2 bytes	<i>att_supreq</i>	Specifies the support requirements.

The symbols are defined in <dnmsdos.h> header file.

B.3 DECnet Node Address Data Structure

The *dn_naddr* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	<i>a_len</i>	Specifies the length of the DECnet node address.
unsigned char	2-byte array	<i>a_addr[DN_MAXADDL]</i>	Specifies the DECnet Phase IV node address for the user task. When <i>a_addr[DN_MAXADDL]</i> is used as a 16-bit unsigned integer, bits 0-9 are the node number, and bits 10-15 are the area number.

The symbols are defined in <dn.h> header file.

B.4 Listen Data Structure

The *listen_dn* data structure contains the following data member:

Type	Size	Data Member	Description
int	2 bytes	<i>lsn_backlog</i>	Defines the maximum number of unaccepted incoming connects which are allowed on this particular socket.

The symbol is defined in <dnmsdos.h> header file.

B.5 Local Node Information Data Structure

The *localinfo_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned char	3-byte array	<i>lcl_version</i>	Specifies the software version number for the network process.
unsigned char	7-byte array	<i>lcl_nodename</i>	Specifies the node name for the local node. It is terminated by a null character.
unsigned short	2 bytes	<i>lcl_nodeaddr</i>	Specifies the DECnet Phase IV node address for the local node. The node address is formatted as a 16-bit unsigned integer, where bits 0-9 are the node number and bits 10-15 are the area number.
unsigned short	2 bytes	<i>lcl_segsize</i>	Specifies the minimum buffer segment size used on the logical link. This number should match the value defined with the NCP command, DEFINE EXECUTOR SEGMENT BUFFER SIZE. (Refer to the <i>DECnet-DOS User's Guide</i> for more details.)
unsigned char	1 byte	<i>lcl_sockets</i>	Specifies the number of sockets available for data exchange.
unsigned char	1 byte	<i>lcl_decnet device</i>	Specifies the DECnet database device name.
exptr	4 bytes	<i>lcl_decnet path</i>	Specifies the address of a buffer that contains the DECnet database path specification string which includes the device name.

The symbols are defined in <dnmsdos.h> header file.

B.6 Logical Link Information Data Structure

The *linkinfo_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	<i>idn_segsize</i>	Specifies the buffer segment size in use on the logical link.
unsigned char	1 byte	<i>idn_linkstate</i>	Specifies the state of the logical link. (See Appendix A for a list of logical link states.)

The symbols are defined in <dn.h> header file.

B.7 Optional User Data Structure

The *optdata_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	<i>opt_status</i>	Specifies an extended status value returned by a function call. A list of the extended error codes appear in Appendix D.
unsigned short	2 bytes	<i>opt_optl</i>	Is the size of the optional user data.
unsigned char	16-byte array	<i>opt_data</i>	Specifies the optional user data.

The symbols are defined in <dn.h> header file.

B.8 Select Data Structure

The *select_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	<i>sel_nfds</i>	Specifies the highest socket number to be checked.
field32	4 bytes	<i>sel_read</i>	Specifies the socket numbers to be examined for read ready or incoming connections.
field32	4 bytes	<i>sel_write</i>	Specifies the socket numbers to be examined for write ready.
field32	4 bytes	<i>sel_except</i>	Specifies the socket numbers to be examined for exception or out-of-band data ready.
unsigned short	2 bytes	<i>sel_seconds</i>	Specifies the time to wait for the socket selection to complete.

NOTE

field32 is the same as unsigned long for type.

The symbols are defined in `<dnmsdos.h>` header file.

B.9 Shutdown Data Structure

The *shutdown_dn* data structure contains the following data member:

Type	Size	Data Member	Description
int	2 bytes	<i>snd_how</i>	Specifies the type of shutdown. The argument can be set to: 0 which disallows further receives. 1 which disallows further sends. 2 which disallows further sends and receives.

The symbol is defined in <dnmsdos.h> header file.

B.10 Socket Address Data Structure

The *sockaddr_dn* data structure contains the following data members:

Type	Size	Data Member	Description
unsigned short	2 bytes	<i>sdn_family</i>	Specifies the communications domain as AF_DECnet.
unsigned char	1 byte	<i>sdn_flags</i>	Specifies the object flag option. It must be set to zero, if not used.
unsigned char	1 byte	<i>sdn_objnum</i>	Defines the object number for the socket.
unsigned short	2 bytes	<i>sdn_objname1</i>	Is the size of the node's object name.
char	16-byte array	<i>sdn_objname</i>	Defines the name of the network task.
struct	4 bytes	<i>sdn_add</i>	Specifies the node address data structure. (See the description of the <i>dn_naddr</i> data structure in this appendix.)

The symbols are defined in <dn.h> header file.

B.11 Socket I/O Status Data Structure

The *sioctl_dn* data structure contains the following data members:

Type	Size	Data Member	Description
int	2 bytes	<i>sio_s</i>	This data member is not used by the <i>attach</i> call.
int	2 bytes	<i>sio_request</i>	Specifies the I/O control level to be used. (See Chapter 4 for details.)
exptr	4 bytes	<i>argp</i>	Specifies the address of the argument list.

The symbols are defined in `<dnmsdos.h>` header file.

B.12 Socket Option Data Structure

The *sockopt_dn* data structure contains the following data members:

Type	Size	Data Member	Description
int	2 bytes	<i>sop_level</i>	<p>Specifies the layer at which options are manipulated.</p> <p>If the level is set to <code>SOL_SOCKET</code>, then the rest of the data structure is ignored. If the level is set to <code>DNPROTO_NSP</code>, then the rest of the data structure can contain either access control and/or optional data or acceptance mode information.</p>
int	2 bytes	<i>sop_optname</i>	<p>Specifies options to be passed for interpretation.</p> <p>When the socket level is set to <code>DNPROTO_NSP</code>, <i>sop_optname</i> can be set to one of 7 specific options. For example, <code>DSO_CONDATA</code>. (See Appendix A for a list of the specific options.)</p>

<i>exptr</i>	4 bytes	<i>sop_optval</i>	Specifies an address for the buffer which contains either access control or optional user data. (See Chapter 4 for the relationship between <i>sop_optname</i> and <i>sop_optval</i> arguments.)
			Specifies an address for the buffer which contains acceptance mode information.
<i>exptr</i>	4 bytes	<i>sop_optlen</i>	Specifies the size of the option value buffer used as a parameter for the <i>setsockopt</i> call.
			It is also a value result parameter for the <i>getsockopt</i> call.

The symbols are defined in <dnmsdos.h> header file.

B.13 User Access Control Information Data Structure

The *dnet_accnt* data structure contains the following data members:

Type	Size	Data Member	Description
char	1 byte	<i>acc_status</i>	Is used internally by this function call.
char	1 byte	<i>acc_type</i>	Specifies the type of privilege associated with the user name or password. The four access types are: 0 - no access rights, 1 - read only access, 2 - write only access, and 3 - read and write access.
char	40-byte array	<i>acc_user</i>	Specifies the user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.
char	40-byte array	<i>acc_pass</i>	Specifies the password associated with a user name. It consists of a 1- to 39-alphabetic character string terminated by a null character.
			These symbols are defined in <dnetdb.h> header file.

B.14 User-Defined Callback Routine Data Structure

The *io_callback* member of the CIOCB has the following format:

Type	Size	Data Member	Description
exptr	4 bytes	<i>io_callback</i>	Specifies the address for the callback routine which will be returned when a function call completes.

You should refer to Chapter 6 of this manual for more details on the CIOCB data structure.

B.15 User-Defined Buffer Data Structure

The *io_buffer* member of the IOCB(CIOCB) data structure has the following format:

Type	Size	Data Member	Description
exptr	4 bytes	<i>io_buffer</i>	Specifies the address for the buffer which contains user defined data.

You should refer to Chapter 6 of this manual for more details on the IOCB and CIOCB data structures.

THE NEW YORK PUBLIC LIBRARY

ASTOR LENOX TILDEN FOUNDATION

155 E. 42ND STREET, NEW YORK 17, N.Y.

DATE TIME

BOOK NO. VOLUME
 AUTHOR TITLE
 SUBJECT

COPIES OF IN
 OF IN

LIBRARY OF THE NEW YORK PUBLIC LIBRARY

ASTOR LENOX TILDEN FOUNDATION

155 E. 42ND STREET, NEW YORK 17, N.Y.

DATE TIME

BOOK NO. VOLUME
 AUTHOR TITLE
 SUBJECT

COPIES OF IN
 OF IN

Summary of Error Completion Codes

This appendix lists the error completion codes returned by DECnet-VAXmate™ in *errno*. They provide extended error information to transparent file access, transparent task-to-task operations, and nontransparent task-to-task communication.

These error codes are a subset of the error codes contained in the external variable *errno*. The following descriptions are standard ULTRIX™ definitions. You should refer to specific DECnet-VAXmate calls for a network definition of the error codes.

Mnemonic	Decimal Value	Description
<hr/> General Network Function I/O Status		
SUCCESS	0	Request has succeeded
ERROR	-1	Request has failed
NOTHING	-2	Request has not completed
 General Error Codes		
EPERM	1	Not owner
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error

ENXIO	6	No such device or address
E2BIG	7	Argument list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No children
EAGAIN	11	No more processes
ENOMEM	12	Not enough core
EACCES	13	Permission was denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY	16	Mount device busy
EEXIST	17	File exists
EXDEV	18	Cross-device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read-only file system
EMLINK	31	Too many links
EPIPE	32	Broken pipe

Math software

EDOM	33	Argument too large
ERANGE	34	Result too large

Nonblocking and Interrupt I/O

EWouldBlock	35	Operation would block
EINPROGRESS	36	Operation now in progress
EALREADY	37	Operation already in progress

Argument errors

ENOTSOCK	38	Socket operation on nonsocket
EDESTADDRREQ	39	Destination address required
EMSGSIZE	40	Message too long
EPROTOTYPE	41	Protocol wrong type for socket
ENOPROTOOPT	42	Protocol not available
EPROTONOSUPPORT	43	Protocol not supported
ESOCKTNOSUPPORT	44	Socket type not supported
EOPNOTSUPP	45	Operation not supported on socket
EPFNOSUPPORT	46	Protocol family not supported
EAFNOSUPPORT	47	Address family not supported by protocol family
EADDRINUSE	48	Address already in use
EADDNOTAVAIL	49	Cannot assign requested address

Operational errors

ENETDOWN	50	Network is down
ENETUNREACH	51	Network is unreachable
ENETRESET	52	Network dropped connection on reset
ECONNABORTED	53	Software caused connection abort

ECONNRESET	54	Connection reset by peer
ENOBUFS	55	No buffer space available
EISCONN	56	Socket is already connected
ENOTCONN	57	Socket is not connected
ESHUTDOWN	58	Cannot send after socket shutdown
ETOOMANYREFS	59	Too many references: cannot splice
ETIMEDOUT	60	Connection timed out
ECONNREFUSED	61	Connection refused
ELOOP	62	Too many levels of symbolic links
ENAMETOOLONG	63	File name too long; should be rearranged
EHOSTDOWN	64	Host is down
EHOSTUNREACH	65	No route to host
ENOTEMPTY	66	Directory not empty

Quotas

EPROCLIM	67	Too many processes
EUSERS	68	Too many users
EDQUOT	69	Disk quota exceeded

D

Summary of Extended Error Codes

DECnet-DOSTM supports extended error support to certain socket operations. When you write a program which uses the *getsockopt* function call, extended error codes can be returned in *opt status*, a data member of *optdata dn*. This can occur following an attempted connection request or after disconnecting a logical link.

Table D-1 lists extended error codes which can be returned following a failed connection. It lists the error messages found in *derrno.h*, the decimal value for each message, the equivalent error message that *dnet_conn* returns in *errno*, and an explanation that indicates why the error was returned.

Table D-1: Extended Error Messages – Unable to Make a Connection

Error Code (Decimal)	derrno.h Mnemonic	dnet_conn In errno	Explanation
0	EREJBYOBJ	ECONNREFUSED	Connection rejected by remote application.
1	EINSSNETRES	ENOSPC	Insufficient network resources.
			Configuration error or transient overload.
2	EUNRNODNAM	EADDRNOTAVAIL	Unrecognized node name. Remote node not defined in local database.

3	EREMNODESHUT	ENETDOWN	Remote node shutting down as a result of remote network management action.
4	EUNROBJ	ESRCH	Unrecognized object. Application not installed at remote node.
5	EINVOBJNAM	EINVAL	Invalid application name format. Local program error.
6	EOJBUSY	ETOOMANYREFS	Object too busy. Application usage exceeds capability.
10	EINVNODNAM	ENAMETOOLONG	Invalid node name format. Local program error.
11	ELOCNODESHUT	EHOSTDOWN	Local node shutting down. Local network management action.
32	ENODERESOURCES	ENOSPC	No resources on either local or remote node for new logical link. Configuration error or transient overload.
33	EUSERRESOURCES	ENOSPC	No remote application resources for new logical link. Application configuration error or overload.
34	EACCONREJ	EACCES	Access control rejected by remote node due to unacceptable access control data.
36	EBADACCOUNT	EACCES	Account information rejected by remote node due to unacceptable access control data.

38	ENORESPOBJ	ETIMEDOUT	No response from remote application.
39	ENODUNREACH	EHOSTUNREACH	Node unreachable. Remote node is down or network is not connected.
43	ECONNTOOBIG	EACCES	Connect data fields too long. Unacceptable data provided.

Table D-2 lists extended error codes which can be returned following a disconnection. It lists the error messages found in *derrno.h*, the decimal value for each message, and the error reason.

Table D-2: Extended Error Messages - Disconnecting a Logical Link

Decimal Error Code	derrno.h Mnemonic	dnet_conn In errno	Description
0	EREJBYOBJ	ECONNREFUSED	The remote application disconnected a running logical link.
8	EABTBYNMGT	ECONNABORTED	Network management disconnected the logical link.
9	EUSERABORT	ECONNRESET	The remote application has aborted the link.
38	ENORESPOBJ	ETIMEDOUT	The remote application has crashed or failed or a network outage has occurred.
39	ENODUNREACH	EHOSTUNREACH	The connection has been lost due to a local timeout.
41	ENOLINK	ENETRESET	No such link found at remote node due to a remote system reboot or multiple nodes in the network.
42	ECOMPLETE	ENOTCONN	Local disconnect successful. Indicates normal condition.

Using the Transparent Network Task Control Utility

You can use the Transparent Network Task (TNT) Control utility to obtain the status of your Transparent File Access (TFA) or Transparent Task-to-Task (TTT) operations. TNT returns DAP messages and other extended error information you can use in troubleshooting your network applications.

This appendix explains how to use TNT to obtain status information about the TFA and TTT utilities. It also describes how to use TNT to remove these utilities from memory.

E.1 Displaying Status

You can run TNT to display status messages for your TFA and/or TTT operations. TNT returns standard MS-DOS[®] error messages in addition to DECnet[™] and Data Access Protocol (DAP) messages. DAP and other extended error messages utilities can help you locate problem areas. See Appendixes F and G for a complete list of these error messages.

To invoke TNT, type the following command:

```
E> TNT Return
```

The system responds with a start-up message and one or more network status messages. For example:

```
Transparent Network Task Control V1.1
```

```
DECnet TFA is not installed.
```

```
DECnet TFA has no errors to report.
```

```
DECnet TFA Errors are: remote file specification: extended error message
```

```
.  
. .  
.
```

or

```
DECnet TTT is not installed.
```

```
DECnet TTT has no errors to report.
```

```
DECnet TTT Errors are: remote file specification : extended error message
```

```
.  
. .  
.
```

where

extended error message returns an error code from one of the following groups of error messages:

- Error codes contained in the external variable *errno* – Appendix C.
- Transparent File Access Routine error messages – Appendix F.
- DAP error messages (maccode/miccode in octal) – Appendix G.

E.2 Removing TFA and TTT From Memory

TFA and TTT are MS-DOS terminate and stay resident tasks that trap MS-DOS interrupt 21H function calls. You can use TNT to remove either TFA or TTT from memory if either one is the last terminate and stay resident task loaded into memory. If other applications that trap INT 21H have been loaded into memory after TFA or TTT, you must remove them before using TNT or reboot your system to remove all memory resident tasks.

To use TNT to remove TFA or TTT from memory, type the TNT OFF command at the DOS prompt. For example:

```
E> TNT TFA OFF Return
```

The system responds with the following text:

```
Transparent Network Task Control V3.0  
The task was removed successfully.
```

If TFA could not be removed, one of the following messages is displayed:

```
Transparent Network Task Control V3.0  
TFA cannot be removed because it is not installed or is not installed last.
```

or if MS-DOS failed on the remove call,

```
Transparent Network Task Control V3.0  
The task could not be removed.
```

E.3 On-Line Help

TNT help provides you with an on-line list of supported TNT commands. To obtain help, type:

```
C> TNT HELP Return
```

The system responds with the following help text:

```
Transparent Network Task Control V3.0  
Transparent Network Task commands are:
```

TNT	Display status of both TTT and TFA.
TNT HELP	Display this text.
TNT TTT OFF	Remove TTT from memory.
TNT TFA OFF	Remove TFA from memory.

If you mistype a command, TNT responds with an error message and the list of supported TNT commands:

```
Transparent Network Task Control V3.0  
Transparent Network Task command error.  
Transparent Network Task commands are:
```

TNT	Display status of both TTT and TFA.
TNT HELP	Display this text.
TNT TTT OFF	Remove TTT from memory.
TNT TFA OFF	Remove TFA from memory.

Transparent File Access Error Messages

This appendix summarizes extended error messages for transparent file access operations. The following messages are displayed by the TNT utility:

Cannot get a handle to the network driver.

Error in closing file.

Error in node specification.

Error – unknown error.

Invalid DAP message format received.

Invalid DAP message type received.

Remote system DAP buffer size is less than 128 bytes.

The buffer size for the records contained in the remote input file is too small.

The file to be accessed is not open.

The maximum record size has exceeded 128 bytes.

The node name specification was not found in DECPARM.DAT.

Too many logical links already in use. The maximum number is 4.

Unable to transmit user buffer.

Unexpected DAP message received.

Unsupported DAP data type.

Unsupported DAP flag field received.

Unsupported file organization. DECnet-DOS supports sequential file organization.

Data Access Protocol (DAP) Error Messages

The Network Task Error log utility provides extended error support to transparent file access operations. This appendix lists DAP error messages that can be returned by this utility. The Network File Transfer utility may also return some of these error messages.

G.1 Overview

The DAP messages return status from the remote file system or from the operation of the cooperating process using DAP. The 2-byte status field (16 bits) is divided into two fields:

Maccode (bits 12–15):

Contains the error type code (see Table G-1).

Miccode (bits 0–11):

Contains the specified error reason code (see Tables G-2, G-3, and G-4, depending on error type).

G.1.1 Maccode Field

The value returned in the maccode field describes the functional type of the error that has occurred. The specific reason for the error is given in the miccode field. Miccode values correlating to each maccode value listed in Table G-1 are found in the table referenced in the last column of Table G-1.

Table G-1: DAP Maccode Field Values

Field Value (Octal)	Error Type	Meaning	Miccode Table
0	Pending	The operation is in progress.	G-3
1	Successful	Returns information that indicates success.	G-3
2	Unsupported	This implementation of DAP does not support the specified request.	G-2
3	Reserved		
4	File open	Errors that occur before a file is successfully opened.	G-3
5	Transfer error	Errors that occur after a file is opened and before it is closed.	G-3
6	Transfer warning	For operations on open files, indicates that the operation completed, but not with complete success.	G-3
7	Access termination	Errors associated with terminating access to a file.	G-3
10	Format	Error in parsing a message. Format is not correct.	G-2
11	Invalid	Field of message is invalid (that is, bits that are meant to be mutually exclusive are set, an undefined bit is set, a field value is out of range, or an illegal string is in a field).	G-2
12	Sync	DAP message received out of synchronization.	G-4
13-15	Reserved		
16-17	User-defined status maccodes		

G.1.2 Miccode Field

The value returned in this field identifies the specific reason for the error type defined in the maccode field (see Section G.1.1). Miccode field values are defined in three different tables, each table associated with certain maccode values, as outlined below:

- Table G-2: For use with maccode values 2, 10, 11
- Table G-3: For use with maccode values 0, 1, 4, 5, 6, 7
- Table G-4: For use with maccode value 12

Table G-2 follows. The DAP message type number (column 1) is specified in bits 6–11, and the DAP message field number (column 2) is specified in bits 0–5. The field where the error is located is described in the third column.

Table G-2: DAP Miccode Values for Use with Maccode Values of 2, 10, 11

Type Number (Bits 6–11)	Field Number (Bits 0–5)	Field Description
Miscellaneous message errors		
00	00	Unspecified DAP message error
	10	DAP message type field (TYPE) error
Configuration message errors		
01	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	BITCNT field (BITCNT)
	20	Buffer size field (BUFSIZ)
	21	Operating system type field (OSTYPE)
	22	File system type field (FILESYS)
	23	DAP version number (VERNUM)
	24	ECO version number field (ECONUM)
	25	USER protocol version number field (USRNUM)
	26	DEC software release number field (DECVER)
	27	User software release number field (USRVER)
	30	System capabilities field (SYSCAP)
Attributes message errors		
02	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN 256)
	14	Bit count field (BITCNT)

Table G-2 (Cont.): DAP Miccode Values for Use with Maccode Values 2, 10, 11

Type Number (Bits 6–11)	Field Number (Bits 0–5)	Field Description
	20	Attributes menu field (ATTMENU)
	21	Data type field (DATATYPE)
	22	Field organization field (ORG)
	23	Record format field (RFM)
	24	Record attributes field (RAT)
	25	Block size field (BLS)
	26	Maximum record size field (MRS)
	27	Allocation quantity field (ALQ)
	30	Bucket size field (BKS)
	31	Fixed control area size field (FSZ)
	32	Maximum record number field (MRN)
	33	Run-time system field (RUNSYS)
	34	Default extension quantity field (DEQ)
	35	File options field (FOP)
	36	Byte size field (BSZ)
	37	Device characteristics field (DEV)
	40	Spooling device characteristics field (SDC); reserved
	41	Longest record length field (LRL)
	42	Highest virtual block allocated field (HBK)
	43	End-of-file block field (EBK)
	44	First free byte field (FFB)
	45	Starting LBN for contiguous file field (SBN)
Access message errors		
03	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Access function field (ACCFUNC)
	21	Access options field (ACCOPT)
	22	File specification field (FILESPEC)
	23	File access field (FAC)
	24	File-sharing field (SHR)
	25	Display attributes request field (DISPLAY)
	26	File password field (PASSWORD)

Table G-2 (Cont.): DAP Miccode Values for Use with Maccode Values 2, 10, 11

Type Number (Bits 6–11)	Field Number (Bits 0–5)	Field Description
Control Message Errors		
04	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Control function field (CTLFUNC)
	21	Control menu field (CTLMENU)
	22	Record access field (RAC)
	23	Key field (KEY)
	24	Key of reference field (KRF)
	25	Record options field (ROP)
	26	Hash code field (HSH); reserved for future use
	27	Display attributes request field (DISPLAY)
	30	Block count (BLKCNT)
Continue message errors		
05	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Continue transfer function field (CONFUNC)
Acknowledge message errors		
06	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	15	System-specific field (SYSPEC)
Access complete message errors		
07	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)

Table G-2 (Cont.): DAP Miccode Values for Use with Maccode Values 2, 10, 11

Type Number (Bits 6–11)	Field Number (Bits 0–5)	Field Description
	20	Access complete function field (CMPFUNC)
	21	File options field (FOP)
	22	Checksum field (CHECK)
Key definition message errors		
12	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Key definition menu field (KEYMENU)
	21	Key option flags field (FLG)
	22	Data bucket fill quantity field (DFL)
	23	Index bucket fill quantity field (IFL)
	24	Key segment repeat count field (SEGCNT)
	25	Key segment position field (POS)
	26	Key segment size field (SIZ)
	27	Key of reference field (REF)
	30	Key name field (KNM)
	31	Null key character field (NUL)
	32	Index area number field (IAN)
	33	Lowest level area number field (LAN)
	34	Data level area number field (DAN)
	35	Key data type field (DTP)
	36	Root VBN for this key field (RVB)
	37	Hash algorithm value field (HAL)
	40	First data bucket VBN field (DVB)
	41	Data bucket size field (DBS)
	42	Index bucket size field (IBS)
	43	Level of root bucket field (LVL)
	44	Total key size field (TKS)
	45	Minimum record size field (MRL)
Allocation message errors		
13	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Allocation menu field (ALLMENU)
	21	Relative volume number field (VOL)

Table G-2 (Cont.): DAP Miccode Values for Use with Maccode Values 2, 10, 11

Type Number (Bits 6–11)	Field Number (Bits 0–5)	Field Description
	22	Alignment options field (ALN)
	23	Allocation options field (AOP)
	24	Starting location field (LOC)
	25	Related file identification field (RFI)
	26	Allocation quantity field (ALQ)
	27	Area identification field (AID)
	30	Bucket size field (BKZ)
	31	Default extension quantity field (DEQ)
Summary message errors		
14	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Summary menu field (SUMENU)
	21	Number of keys field (NOK)
	22	Number of areas field (NOA)
	23	Number of record descriptors field (NOR)
	24	Prologue version number (PVN)
Date and time message errors		
15	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Date and time menu field (DATMENU)
	21	Creation date and time field (CDT)
	22	Last update date and time field (RDT)
	23	Deletion date and time field (EDT)
	24	Revision number field (RVN)
	25	Backup date and time field (BDT)
	26	Physical creation date and time field (PDT)
	27	Accessed date and time field (ADT)
Protection message errors		
16	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)

Table G-2 (Cont.): DAP Miccode Values for Use with Maccode Values 2, 10, 11

Type Number (Bits 6–11)	Field Number (Bits 0–5)	Field Description
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Protection menu field (PROTMENU)
	21	File owner field (OWNER)
	22	System protection field (PROTSYS)
	23	Owner protection field (PROTOWN)
	24	Group protection field (PROTGRP)
	25	World protection field (PROWLD)
Name message errors		
17	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	20	Name type field (NAMETYPE)
	21	Name field (NAMESPEC)
Access control list message errors (reserved for future use)		
20	00	Unknown field
	10	DAP message flags field (FLAGS)
	11	Data stream identification field (STREAMID)
	12	Length field (LENGTH)
	13	Length extension field (LEN256)
	14	Bit count field (BITCNT)
	15	System-specific field (SYSPEC)
	20	Access control list repeat count field (ACLCNT)
	21	Access control list entry field (ACL)

Table G-3 follows. The error code number (column 1) is contained in bits 0–11. For corresponding RMS or FCS status codes, refer to the appropriate DECnet or RMS documentation for each remote system.

Table G-3: DAP Miccode Values for Use with Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
0	Unspecified error
1	Operation aborted
2	F11-ACP could not access file
3	File activity precludes operation
4	Bad area ID
5	Alignment options error
6	Allocation quantity too large or 0 value
7	Not ANSI D format
10	Allocation options error
11	Invalid (that is, synchronous) operation at AST level
12	Attribute read error
13	Attribute write error
14	Bucket size too large
15	Bucket size too large
16	BLN length error
17	Beginning of file detected
20	Private pool address
21	Private pool size
22	Internal RMS error condition detected
23	Cannot connect RAB
24	\$UPDATE changed a key without having attribute of XB\$CHG set
25	Bucket format check-byte failure
26	RSTS/E close function failed
27	Invalid or unsupported COD field
30	F11-ACP could not create file (STV – system error code)

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0-11)	Error Description
31	No current record (operation not preceded by get/find)
32	F11-ACP deaccess error during close
33	Data area number invalid
34	RFA-accessed record was deleted
35	Bad device, or inappropriate device type
36	Error in directory name
37	Dynamic memory exhausted
40	Directory not found
41	Device not ready
42	Device has positioning error
43	DTP field invalid
44	Duplicate key detected; XB\$DUP not set
45	F11-ACP enter function failed
46	Operation not selected in ORG\$ macro
47	End of file
50	Expanded string area too short
51	File expiration date not yet reached
52	File extend failure
53	Not a valid FAB (BID does not = FB\$BID)
54	Illegal FAC for record operation, or FB\$PUT not set for create
55	File already exists
56	Invalid file ID
57	Invalid flag-bits combination
60	File is locked by other user
61	F11-ACP find function failed
62	File not found
63	Error in file name
64	Invalid file options
65	Device/file full

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
66	Index area number invalid
67	Invalid IFI value or unopened file
70	Maximum NUM (254) areas/key XABS exceeded
71	\$INIT macro never issued
72	Operation illegal or invalid for file organization
73	Illegal record encountered (with sequential files only)
74	Invalid ISI value on unconnected RAB
75	Bad key buffer address (KBF = 0)
76	Invalid key field (KEY = 0 or negative)
77	Invalid key of reference (\$GET/\$FIND)
100	Key size too large
101	Lowest level index area number invalid
102	Not ANSI-labeled tape
103	Logical channel busy
104	Logical channel number too large
105	Logical extend error; prior extend still valid
106	LOC field invalid
107	Buffer-mapping error
110	F11-ACP could not mark file for deletion
111	MRN value = negative or relative key > MRN
112	MRS value = 0 for fixed length records and/or relative files
113	NAM block address invalid (NAM = 0 or is not accessible)
114	Not positioned to EOF (with sequential files only)
115	Cannot allocate internal index descriptor
116	Indexed file; primary key defined
117	RSTS/E open function failed
120	XABs not in correct order
121	Invalid file organization value
122	Error in file's prolog (reconstruct file)

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
123	POS field invalid (POS > MRS; STV = XAB indicator)
124	Bad file date field retrieved
125	Privilege violation (OS denies access)
126	Not a valid RAB (BID does not = RB\$BID)
127	Illegal RAC value
130	Illegal record attributes
131	Invalid record buffer address (either odd or not word aligned if BLK–IO)
132	File read error
133	Record already exists
134	Bad RFA value (RFA=0)
135	Invalid record format
136	Target bucket locked by another stream
137	F11–ACP remove function failed
140	Record not found
141	Record not locked
142	Invalid record options
143	Error while reading prolog
144	Invalid RRV record encountered
145	RAB stream currently active
146	Bad record size (RSZ > MRS or NOT = MRS if fixed length records)
147	Record too big for user's buffer
150	Primary key out of sequence (RAC = RB\$SEQ for \$PUT)
151	SHR field invalid for file (cannot share sequential files)
152	SIZ field invalid
153	Stack too big for save area
154	System directive error
155	Index tree error

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
156	Error in file type extension on FNS is too big
157	Invalid user buffer address (0, odd, or not word aligned if BLK–IO)
160	Invalid user buffer size (USZ=0)
161	Error in version number
162	Invalid volume number
163	File write error (STV = system error code)
164	Device is write locked
165	Error while writing prolog
166	Not a valid XAB (@XAB = odd; STV = XAB indicator)
167	Default directory invalid
170	Cannot access argument list
171	Cannot close file
172	Cannot deliver AST
173	Channel assignment failure (STV = system error code)
174	Terminal output ignored due to <CTRL/O>
175	Terminal input aborted due to <CTRL/Y>
176	Default file name string address error
177	Invalid device ID field
200	Expanded string address error
201	File name string address error
202	FSZ field invalid
203	Invalid argument list
204	Known file found
205	Logical name error
206	Node name error
207	Operation successful
210	Inserted record had duplicate key
211	Index update error occurred; record inserted

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
212	Record locked, but read anyway
213	Record inserted in primary key is okay; may not be accessible by secondary keys or RFA
214	File was created, but not opened
215	Bad prompt buffer address
216	Asynchronous operation pending completion
217	Quoted string error
220	Record header buffer invalid
221	Invalid related file
222	Invalid resultant string size
223	Invalid resultant string address
224	Operation not sequential
225	Operation successful
226	Created file superseded existing version
227	File name syntax error
230	Timeout period expired
231	FB\$BLK record attribute not supported
232	Bad byte size
233	Cannot disconnect RAB
234	Cannot get JFN for file
235	Cannot open file
236	Bad JFN value
237	Cannot position to end of file
240	Cannot truncate file
241	File currently in an undefined state; access is denied
242	File must be opened for exclusive access
243	Directory full
244	Handler not in system
245	Fatal hardware error

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
246	Attempt to write beyond EOF
247	Hardware option not present
250	Device not attached
251	Device already attached
252	Device not attachable
253	Shareable resource in use
254	Illegal overlay request
255	Block check or CRC error
256	Caller's nodes exhausted
257	Index file full
260	File header full
261	Accessed for write
262	File header checksum failure
263	Attribute control list error
264	File already accessed on LUN
265	Bad tape format
266	Illegal operation on file descriptor block
267	Rename; two different devices
270	Rename; new file name already in use
271	Cannot rename old file system
272	File already open
273	Parity error on device
274	End of volume detected
275	Data overrun
276	Bad block on device
277	End of tape detected
300	No buffer space for file
301	File exceeds allocated space; no blocks left
302	Specified task not installed

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
303	Unlock error
304	No file accessed on LUN
305	Send/receive failure
306	Spool or submit command file failure
307	No more files
310	DAP file transfer checksum error
311	Quota exceeded
312	Internal network error condition detected
313	Terminal input aborted due to <CTRL/C>
314	Data bucket fill size > bucket size in XAB
315	Invalid expanded string length
316	Illegal bucket format
317	Bucket size of LAN does not = IAN in XAB
320	Index not initialized
321	Illegal file attributes (corrupt file header)
322	Index bucket fill size > bucket size in XAB
323	Key name buffer not readable or writeable in XAB
324	Index bucket will not hold two keys for key of reference
325	Multibuffer count invalid (negative value)
326	Network operation failed at remote node
327	Record is already locked
330	Deleted record successfully accessed
331	Retrieved record exceeds specified key value
332	Key XAB not filled in
333	Nonexistent record successfully accessed
334	Unsupported prolog version
335	Illegal key of reference in XAB
336	Invalid resultant string length
337	Error updating RRVs; some paths to data may be lost

Table G-3 (Cont.): DAP Miccode Values for Maccode Values 0, 1, 4, 5, 6, 7

Error Code (Bits 0–11)	Error Description
340	Data types other than string limited to one segment in XAB
341	Reserved
342	Operation not supported over network
343	Error on write behind
344	Invalid wildcard operation
345	Working set full (cannot lock buffers in working set)
346	Directory listing: error in reading volume set name, directory name, or file name
347	Directory listing: error in reading file attributes
350	Directory listing: protection violation in trying to read the volume set, directory, or file name
351	Directory listing: protection violation in trying to read file attributes
352	Directory listing: file attributes do not exist
353	Directory listing: unable to recover directory list after continue transfer (skip)
354	Sharing not enabled
355	Sharing page count exceeded
356	UPI bit not set when sharing with BRO set
357	Error in access control string
360	Terminator not seen
361	Bad escape sequence
362	Partial escape sequence
363	Invalid wildcard context value
364	Invalid directory rename operation
365	User structure (FAB/RAB) became invalid during operation
366	Network file transfer mode precludes operation
6000 to 7777	User-defined errors

Table G-4 follows. The message type number is contained in bits 0–11.

Table G-4: DAP Miccode Values for Use with Maccode Value 12

Type Number (Bits 0–11)	Message Type
0	Unknown message type
1	Configuration message
2	Attributes message
3	Access message
4	Control message
5	Continue transfer message
6	Acknowledge message
7	Access complete message
10	Data message
11	Status message
12	Key definition attributes extension message
13	Allocation attributes extension message
14	Summary attributes extension message
15	Date and time attributes extension message
16	Protection attributes extension message
17	Name message
20	Access control list extended attributes message

Transporting DECnet-DOS Programs

If you develop code to be transported to a DECnet-ULTRIX™ or any other system that supports the socket interface library, it is recommended that you make note of these suggestions:

- The *select* function has a feature specific only to DECnet-VAXmate. The bit mask *exceptfds* is presently not supported by DECnet-ULTRIX. As a result, you cannot transport code that includes the exception bit mask.
- Include a special prefix and compatibility mode header file in your DECnet-VAXmate program.
- Define certain function call names depending upon which system compilation is to take place. For DECnet-VAXmate programs, the socket function calls – *ioctl*, *read*, *write* and *close* must be prefixed with an “s.”

An example compatibility header file is shown here:

```
#ifdef MSDOS

#define ioctl(s, f, a)      sioctl(s, f, a)      /* control */
                                           /* socket i/o*/

#define read(s, buf, len)  sread(s, buf, len)    /* read from */
                                           /* a socket */
#define write(s, buf, len) swrite(s, buf, len)   /* write to */
                                           /* a socket */
#define close(s)           sclose(s)             /* close a */
                                           /* socket */
#endif
                                           /* MSDOS */
```

H

Reporting OGC-102 Programs

The following information is required for the reporting of OGC-102 programs. The information should be provided in the following format:

1. Program Name: [Name of the program]

2. Program Description: [Brief description of the program]

3. Program Objectives: [List of program objectives]

4. Program Results: [Summary of program results]

5. Program Evaluation: [Summary of program evaluation]

6. Program Funding: [Summary of program funding]

7. Program Contact: [Contact information for the program]

8. Program Status: [Current status of the program]

9. Program Notes: [Any additional notes or comments]

DECnet-DOS Programming Examples

I.1 Example Client Task Program

The following networking program uses the DECnet-DOS™ socket interface. In this example, the client task tries to connect to a task on a remote node and then tries to send data and waits 30 seconds for any incoming data before timing out or until the peer task decides to close down the link.

```
/*
 * Include standard headers.
 */
#include <stdio.h>

/*
 * User-defined symbols for conditional compilation.
 */
#include "dnprefix.h"

/*
 * Include some network interface headers.
 */
#include "types.h" /* Type definitions, abstract data types. */
#include "time.h" /* Time data structures. */
#include "dn.h" /* Network data structures and definitions. */

/*
#include "socket.h" /* Socket interface layer definitions. */
#include "siocctl.h" /* Socket I/O control functions. */
#include "errno.h" /* Global user error definitions returned
                  /* in 'errno'.
*/

/*
 * Conditionalize for DECnet-ULTRIX compatibility.
 */
#ifdef MSDOS
#define sclose(s) close(s)
```

```

#define ioctl(s,f,a) ioctl(s,f,a)
#endif

#define SEQUENCED_PACKET 0
#define STREAM          1

/*
 * Version string.
 */
static char      version[] = "v1.01";

/*
 * Main line code.
 */
main(argc, argv)

int      argc;
char     *argv[];
{
    /*
     * Local data.
     */
    struct timeval tmv;
    char      *node, *object;
    u_char    optional_send[16];
    u_char    optional_receive[16];
    u_char    data_buffer[10];
    field32    readfds, writefds;
    int        rec_len;
    int        sock_type;
    int        sock;
    int        loop;
    int        count;
    int        len;
    int        ind0;
    char       bio[1];

    /*
     * Make sure there are a valid number of input arguments.
     */
    if (argc < 3)
    {
        printf("Usage: test <node name or address>\n
        <#objnum or objnam>\n");
        exit(1);
    }
    /*
     * Display our current version.
     */
    printf("\t\tSample - %s\n", &version [0]);

    /*
     * Set up optional data to send with connect.
     */
    strcpy(&optional_send[0], "hello");

```

```

/*
 * Attempt to connect to the object on the remote node.
 */
rec_len = sizeof(optional_receive);
node = *++argv;
object = *++argv;
sock_type = SEQUENCED_PACKET;
printf("connecting to node \"%s\", object \"%s\"\n",
       node, object);
if ((sock = dnet_conn(node, object, sock_type,
                     &optional_send[0],
                     strlen("hello"),
                     &optional_receive[0], &rec_len)) < 0)
{
    perror("dnet_conn");
    exit(1);
}

printf("connect complete with node \"%s\", \
object \"%s\"\n", node, object);

/*
 * Check for returned optional data.
 */
if (rec_len)
{
    puts("optional data received:");
    for (ind0 = 0; ind0 < rec_len; ind0++)
    {
        printf(" %d", optional_receive[ind0]);
    }
    puts("");
}

/*
 * Fill a data buffer with dummy data.
 */
for (loop = 0; loop < sizeof(data_buffer); loop++)
{
    data_buffer[loop] = loop;
}

/*
 * Try to send a dummy data buffer 10 times
 * to target object as long as link is still active.
 */
loop = 10;
while (loop--)
{
    if (dnet_eof(sock) == 1)
    {
        printf("link is down.\n");
        sclose(sock);
        exit(1);
    }
}

```



```

    }

    if ((count = send(sock, &data_buffer[0],
                      sizeof(data_buffer), 0)) < 0)
    {
        perror("write");
        sclose(sock);
        exit(1);
    }
}
printf("data successfully sent to %s\n", node);

/*
 * Now set the socket to nonblocking mode.
 */
bio[0] = 1;
ioctl(sock, FIONBIO, &bio[0]);

/*
 * Clean out the data buffer.
 */
bzero(&data_buffer[0], sizeof(data_buffer));

/*
 * Continue to receive data from target object until
 * disconnected.
 */
while (1)
{
    /*
     * Check if link is still active.
     */
    if (dnet_eof(sock) == 1)
    {
        printf("link is down.\n");
        sclose(sock);
        exit(1);
    }

    /*
     * Now check to see if the socket has data available
     * to read and timeout after 30 seconds.
     */
    readfds = 1 << sock;
    tmv.tv_sec = 30;
    if ((ind0 = select(sock + 1, &readfds, 0, 0, &tmv)) < 0)
    {
        perror("select");
    }
    else
    {
        if (ind0 == 0)
        {
            printf("receive wait timed out.\n");

```

```

        sclose(sock);
        exit(1);
    }
}

if ((count = recv(sock, &data_buffer[0],
                  sizeof(data_buffer), 0)) < 0)
{
    if (errno != EWOULDBLOCK)
    {
        perror("read");
        break;
    }
    else
        continue;
}
printf("data received (%d bytes):\n", count);
for (ind0 = 0; ind0 < count; ind0++)
{
    printf(" %d", data_buffer[ind0] );
}
puts("");
}

/*
 * Finish up. Make the socket linger on close to allow
 * things to get cleaned.
 */
if (setsockopt(sock, SOL_SOCKET, SO_LINGER, 0, 0) < 0)
{
    perror("setsockopt");
}

/*
 * Close the socket and exit program.
 */
sclose(sock);
exit(0);
}

```

1.2 Example Client Transparent Task-to-Task Program

The following program illustrates transparent task-to-task communication. It describes the functions that the client task uses to communicate over the network.

```
/*
 *
 * Sample client program written in C that shows Transparent
 * task-to-task using DECnet-DOS.
 *
 * When running this program, the command line argument should
 * look like a network task specification. See the following
 * examples as well as examples cited in the documentation:
 * For example:
 *
 *      \\t\pcdos\\#100(to connect by object number)
 *      \\t\pcdos\smith\xxxxx\\TIMESRV(to connect by object name)
 *
 * After getting a handle (for example, by connecting to a
 * remote object), an attempt is made to write/send some data to
 * the object and then close the handle.
 *
 * o All C include files and external functions are
 *   distributed with the DECnet-DOS kit in the file
 *   DNETLIB.SRC.
 *
 * o When attempts to run this program fail, run the utility
 *   TNT.EXE shipped with the DECnet-DOS kit to examine
 *   DECnet errors.
 */
#include      <stdio.h>
#include      "types.h"
#include      "scbdef.h"
#include      "errno.h"

static char buf[100];

/*
 * Function(s) included in DNETLIB.SRC
 */
extern int hopen();
extern int hwrite(), hclose();

main(argc, argv)

int argc;

char *argv[];
```



```

{
    int i;
    int j;
    int len;
    int h = 0;

    if (argc < 2)
    {

        printf("Usage: tttst <TTT_network_task_string>\n");
        printf("\n example:\n");
        printf("\t tttst \\\t\\pygmy\\\t<#object_number>\n");
        printf("\t or\n");
        printf("\t tttst \\\t\\pygmy\\\t<object_name>\n");
        exit(1);
    }

    /*
     * Fill a dummy data buffer.
     */
    for (i = 0; i < sizeof(buf); i++)
        buf[i] = i;

    /*
     * Open file (access remote network object).
     */
    h = hopen(argv[1], SCBC_HOPEN);
    if (h == ERROR)
    {
        perror("\nopen");
        printf("\n (run TNT.EXE to examine network error)");
        exit(1);
    }
    printf("\nopen succeeded handle: %u (connected to object)",
        h);

    /*
     * Write to file (send data to remote object).
     */
    if (fwrite(h, &buf[0], sizeof(buf)) != sizeof(buf))
    {
        perror("\nwrite");
        printf("\n (run TNT.EXE to examine network error)");
    }
    else
        printf("\nwrite succeeded (sent data to object)");
}

```

```

/*
 * Read from file handle (receive data from object, if any).
 */
len = hread(h, &buf[0], sizeof(buf));

if (len < 0)
{
    perror("\nread");
    printf("\n (run TNT.EXE to examine network error)");
}
else
{
    printf("\nread %u byte(s) (received from object)\n",
        len);
    for (i = j = 0; i < len; i++, j++)
    {
        if (j > 9)
        {
            printf("\n");
            j = 0;
        }
        printf(" %4u", buf[i]);
    }
}

/*
 * Close file handle (disconnect link).
 */
hclose(h);

printf("\nfinished.");
exit(0);
}

```

1.3 Example Server Task Program

The following networking program uses the DECnet-DOS socket interface. The example describes the activities of a DECnet-DOS server task.

```
/*
 * Program - MIRROR
 *
 * Copyright (C) 1985, All Rights Reserved, by
 * Digital Equipment Corporation, Maynard, Mass.
 *
 * This software is furnished under a license and may be used
 * and copied only in accordance with the terms of such license
 * and with the inclusion of the above copyright notice. This
 * software or any other copies thereof may not be provided or
 * otherwise made available to any other person. No title to
 * and ownership of the software is hereby transferred.
 *
 * The information in this software is subject to change without
 * notice and should not be construed as a commitment by
 * Digital Equipment Corporation.
 *
 * Digital assumes no responsibility for the use or reliability
 * of its software on equipment which is not supplied
 * by Digital.
 *
 * MODULE DESCRIPTION:
 *
 * Program MIRROR
 *
 * DECnet-DOS, mirror server, DECnet object 25
 *
 * Networks & Communications Software Engineering
 *
 * IDENT HISTORY:
 *
 * V1.00          20-Nov-85
 *                DECnet-DOS, Version 1.1
 */

#include <stdio.h>
#include "types.h"
#include "dnmsdos.h"
#include "dn.h"
#include "socket.h"
#include "time.h"
#include "errno.h"

#include "scbdef.h"
```



```

#define MAX_BUF_SIZE 2048      /* maximum loop data buffer */

struct sockaddr_dn sockaddr; /* accept connect data structure */
struct optdata_dn opt;      /* optional data buffer */
char buff[MAX_BUF_SIZE];    /* data buffer */
int lsock = -1;              /* incoming connections on */
                              /* listening socket */

int sock = -1;               /* data communications socket */
char mode[1];                /* accept mode */
char msg_version[] = "MIRROR listening (V1.1)";

/*
 * Sample DECnet-DOS server task. This task will bind itself
 * as DECnet object number 25, the standard DECnet object
 * reserved for a mirror task. When started, the mirror is the
 * only running task. To terminate, the user may * press any key.
 */
main(argc, argv)
int argc;
char **argv;
{
    extern char *malloc();
    extern char *dnet_ntoa();
    int len;
    int nfds;
    unsigned long read;
    struct timeval tmv;

    /*
     * Set up listening socket for incoming connect requests.
     */
    if ((lsock = socket(AF_DECnet, SOCK_SEQPACKET, 0)) < 0)
        mir_exit("socket failed", errno);

    /*
     * Bind task to DECnet object 25.
     */
    bzero(&sockaddr, sizeof(sockaddr));
    sockaddr.sdn_family = AF_DECnet;
    sockaddr.sdn_objnum = 25;
    if (bind(lsock, &sockaddr, sizeof(sockaddr)) < 0)
        mir_exit("bind failed", errno);

    /*
     * Set up listening socket to listen for incoming connect
     * requests. Allow for up to 5 pending incoming * connect requests.
     */
    if (listen(lsock, 5) < 0)
        mir_exit("listen failed", errno);

    /*
     * Listen for incoming connect requests until
     * there is keyboard input.
     */
}

```

```

while(1)
{
    /*
     * Display mirror version message.
     */
    printf("\n%s", msg_version);

    /*
     * Poll listening socket for incoming connect request.
     */
    while(1)
    {
        if (mir_keyboard_input())
            mir_exit(NULL, 0);
        bzero(&tmv, sizeof(tmv));
        read = 1 << lsock;
        nfds = lsock + 1;
        if (select(nfds, &read, 0, 0, &tmv) > 0)
        {
            if (read & (1 << lsock))
                break;
        }
    }

    /*
     * Issue a deferred accept on the connect request - send
     * some optional data along with it.
     */
    mode[0] = ACC_DEFER;
    if (setsockopt(lsock, DNPROTO_NSP, DSO_ACCEPTMODE,
        &mode[0], sizeof(mode)) < 0)
    {
        mir_exit("set accept mode", 1);
    }

    len = sizeof(sockaddr);
    if ((sock = accept(lsock, &sockaddr, &len)) < 0)
        mir_exit("accept failed", errno);

    /*
     * Set up outgoing optional data - maximum mirror
     * data buffer size.
     */
    bzero(&opt, sizeof(opt));
    opt.opt_opt1 = sizeof(unsigned short);

    *(unsigned short *)&opt.opt_data[0] = MAX_BUF_SIZE;
    if (setsockopt(sock, DNPROTO_NSP, DSO_CONDATA, &opt,
        sizeof(opt)) < 0)
    {
        mir_exit("set socket option - optional data",
            errno);
    }
}

```

```

if (setsockopt(sock, DNETPROTO_NSP, DSO_CONACCEPT,
0, 0) < 0)
{
    mir_exit("set connect accept", 1);
}

/*
 * Display peer information.
 */
printf("\n");
printf("\nLoop connect request from node: %s",
    dnet_ntoa(&sockaddr.sdn_add));

if (sockaddr.sdn_objnum == 0)
    printf("\nRequesting object name: %s",
        &sockaddr.sdn_objname[0]);

else
    printf("\nRequesting object number: %d",
        sockaddr.sdn_objnum);
printf("\n");

/*
 * Loop data while link is still active and other end is
 * still sending data.
 */
while(!dnet_eof(sock))
{
    len = MAX_BUF_SIZE;
    len = sread(sock, buff, &len);
    if (len == 0)
    {
        if (dnet_eof(sock))
            mir_exit(NULL, 0);
    }
    else
    {
        if (len < 0)
            mir_exit("sread", 1);
    }

    if (buff[0] != 0)
    {
        buff[0] = -1;
        len = 1;
    }
    else
    {
        buff[0] = 1;
    }

    if (swrite(sock, buff, len) < 0)
        mir_exit("swrite", 1);
}

```



```

        /*
        * Finished with current data socket, close it up.
        */
        if (sock != -1)
            sclose(sock);
    }
}

int mir_keyboard_input()
{
    SCB scb;
    scb.AH = SCBC_CKSTAT;
    msdos(&scb);
    if (scb.AL)
        return(1);
    return(0);
}

mir_exit(sp, err)
char *sp;
int err;
{
    if (sp != NULL)
    {
        strcpy(buff, "\\nmirror - ");
        strcat(buff, sp);
        perror(buff);
    }

    if (lsock != -1)
        sclose(lsock);

    if (sock != -1)
        sclose(sock);
    exit(err);
}

```

1.4 Example Client Task-to-Task Program

The following program is an example of a client task loop test. You can use this program to attempt a connection to any remote DECnet object.

```
/*
 * Sample Client Program - LOOP.C
 *
 *   o DECnet-DOS loop test example client task.
 *   o Can be used to attempt to connect to any remote DECnet
 *     object (works best with DECnet MIRROR object, #25).
 *   o Should be linked with a DNET.LIB DECnet-DOS C programming
 *     library (ie: built from sources from DNETLIB.SRC).
 */

#include <stdio.h>

/*
 * . User-defined symbols for conditional compilation.
 * . Not part of standard Ultrix environment.
 * . Copy this file to the default compilation directory on the
 * Ultrix system and 'undef'ine the MSDOS conditional.
 */
#include "dnprefix.h"

/*
 * Include some network interface headers.
 */
#ifdef MSDOS

#include "/usr/sys/h/types.h" /* Type definitions, abstract data types. */
#include "/usr/include/time.h" /* Time data structures. */
#include "/sys/netdnet/dn.h" /* Network data structures and definitions. */
#include "/usr/sys/h/socket.h" /* Socket interface layer definitions. */
#include "/usr/sys/h/ioctl.h" /* Socket I/O control functions. */
#include "/usr/include/errno.h" /* Global user error definitions returned
in 'errno'. */

#else

#include "types.h" /* Type definitions, abstract data types. */
#include "time.h" /* Time data structures. */
#include "dn.h" /* Network data structures and definitions. */
#include "socket.h" /* Socket interface layer definitions. */
#include "siioctl.h" /* Socket I/O control functions. */
#include "errno.h" /* Global user error definitions returned in 'errno'. */

#endif
```

```

/*
 * Conditionalize for DECnet-ULTRIX compatibility.
 */
#ifdef MSDOS
#define sclose(s) close(s)
#define sread(s,b,l) read(s,b,l)
#define swrite(s,b,l) write(s,b,l)
#define siocctl(s,f,a) ioctl(s,f,a)
#endif

#define SEQUENCED_PACKET 0
#define STREAM 1

/*
 * Use for wrapping data display to a new line.
 */
#define NEW_LINE 10

/*
 * Version string.
 */
static char loop_version[] = "V1.2.8 (01-Aug-1987)";

/*
 * Data buffers
 */
u_char data
u_char data_buffer[128];
u_char oob_data_buffer[16];
u_char optional_send[16];
u_char optional_receive[16];

/*
 * More local data
 */

#ifdef MSDOS
int readfds, exceptfds;
#else
long readfds, exceptfds;
#endif

struct timeval tmv;
char *node, *object;
int sock = -1; /* Set to ERROR (-1) initially */
int new_line;
int rec_len;
int sock_type;
int iterations;
int count;
int len;
int ind0;
char bio[1];

```



```

/*
 * Main line code.
 */
main(argc, argv)
int    argc;
char   *argv[];
{

    /*
     * Display our current version.
     */
    printf("\tLOOP - Example Client Program - %s\n",
           &loop_version[0]);

    /*
     * Make sure there are a valid number of input arguments.
     */
    if (argc < 3)
    {
        printf("Usage: loop <node> <object> [timeout] [count]\n");
        printf("  where:\n");
        printf("  node -> target node name or address\n");
        printf("  object -> target object name or number\n");
        printf("  timeout -> timeout value in seconds (optional, default:15)\n");
        printf("  count -> number of messages to send (optional, default:10)\n");
        printf("  \nExample:\n");
        printf("  loop pcdos #25 30 100\n");
        loop_exit(1, sock);
    }

    /*
     * Set up optional data to send with connect;
     * set it to the value of the length of our data buffer,
     * in bytes.
     */
    *(unsigned short *)&optional_send[0] = sizeof(data_buffer);
    rec_len = sizeof(optional_receive);

    /*
     * Get required command line arguments.
     */
    node = *++argv;
    object = *++argv;

    /*
     * Set up optional command line arguments.
     */
    tmv.tv_sec = 15L;
    if (argc > 3)
        tmv.tv_sec = (unsigned long)atoi(*++argv);

```

```

iterations = 10;
if (argc > 4)
    iterations = atoi(++argv);

/*
 * Attempt to connect to the object on the remote node.
 */
sock_type = SEQUENCED_PACKET;
printf("Connecting to node \"%s\", object \"%s\"\n",
       node, object);

if ((sock = dnet_conn(node,
                      object,
                      sock_type,
                      &optional_send[0],
                      sizeof(unsigned_short),
                      &optional_receive[0],
                      &rec_len)) < 0)
{
    perror("dnet_conn");
    loop_exit(1, sock);
} printf("\nConnect complete with node \"%s\", object \"%s\" (socket %u)\n",
       node, object, sock);

/*
 * Check for returned optional data.
 */
if (rec_len)
{
    printf("Optional data received:\n");
    for (ind0 = 0; ind0 < rec_len; ind0++)
        printf(" %d", optional_receive[ind0]);
    puts("");
}

/*
 * Fill a data buffer with dummy data.
 */
for (ind0 = 0; ind0 < sizeof(data_buffer); ind0++)
    data_buffer[ind0] = ind0;

/*
 * Try to send a dummy data buffer for the specified number of iterations
 * to target object as long as link is still active.
 */
while (iterations--)
{
    if (dnet_eof(sock) == 1)
    {
        printf("Link (%u) is down\n", sock);
        loop_exit(1, sock);
    }
}

```

```

        if ((count = send(sock, &data_buffer[0],
                           sizeof(data_buffer), 0)) < 0)
        {
            perror("Send");
            loop_exit(1, sock);
        }
    }
    printf("Data successfully sent to %s\n", node);

    /*
     * Now set the socket to nonblocking mode.
     */
    bio[0] = 1;
    ioctl(sock, FIONBIO, &bio[0]);

    /*
     * Clean out the data buffer.
     */
    bzero(&data_buffer[0], sizeof(data_buffer));

    /*
     * Continue to receive data from target object until * disconnected.
     */
    for (;;)
    {
        /*
         * Check if link is still active.
         */
        if (dnet_eof(sock) == 1)
        {
            printf("Link (%u) is down\n", sock);
            loop_exit(1, sock);
        }

        /*
         * Now check to see if the socket has data available
         * to read and timeout after specified seconds.
         */
        exceptfds = readfds = 1 << sock;

#ifdef DEBUG
        printf("\nCalling select()...\n");
        printf(" timeout set to: %lu\n", tmv.tv_sec);
        printf(" read mask set to: %lxH\n", readfds);
        printf(" except mask set to: %lxH\n", exceptfds);
#endif
#ifdef MSDOS
        printf("Press <Space Bar> to continue...");
        pausec(' ');
#endif
        puts(" ");
    }
#endif

```



```

        if ((ind0 = select(sock + 1,
                           &readfds,
                           NULL,
                           &exceptfds,
                           &tmv)) < 0)
        {
            perror("Select");
        }
        else
        {
            if (ind0 == 0)
            {
                printf("Receive wait (%lu seconds) timed out on link %u\n",
                       tmv.tv_sec, sock);
                loop_exit(1, sock);
            }
        }

#ifdef DEBUG
        printf("\nReturning from select(), status: %d\n", ind0);
        printf(" read mask return value: %lxH\n", readfds);
        printf(" except mask return value: %lxH\n", exceptfds);
#endif
#ifdef MSDOS
        printf("Press <Space Bar> to continue...");
        pausec(' ');
#endif
        puts("");
#endif

        /*
         * Do we have any more data to read off of socket?
         */
        if (!(readfds & (1 << sock)) && !(exceptfds & (1 << sock)))
        {
            printf("\nNo more receive data...\n");
            break;
        }

        if (readfds & (1 << sock))
        {
            if ((count = sread(sock, &data_buffer[0], sizeof(data_buffer))) < 0)
            {
                if (errno != EWOULDBLOCK)
                {
                    perror("Sread");
                    break;
                }
                else
                    continue;
            }
            printf("\nData received (%d bytes):\n", count);
            for (new_line = ind0 = 0; ind0 < count; ind0++, new_line++)
            {

```

```

        if (new_line > NEW_LINE)
        {
            puts("");
            new_line = 0;
        }
        printf(" %4d", data_buffer[ind0]);
    }
    puts("");
}

if (exceptfds & (1 << sock))
{
    if ((count = recv(sock, &oob_data_buffer[0],
                     sizeof(oob_data_buffer), MSG_OOB)) < 0)
    {
        if (errno != EWOULDBLOCK)
        {
            perror("Receive out-of-band");
            break;
        }
        else
            continue;
    }

    printf("\nOut-of-band data received (%d bytes):\n", count);
    for (new_line = ind0 = 0; ind0 < count; ind0++, new_line++)
    {
        if (new_line > NEW_LINE)
        {
            puts("");
            new_line = 0;
        }
        printf(" %4d", oob_data_buffer[ind0]);
    }
    puts("");
}

}

/*
 * Finish up. Make the socket linger on close to allow
 * things to get cleaned.
 */
printf("Setting socket option to linger upon closing...\n");
if (setsockopt(sock, SOL_SOCKET, SO_LINGER, NULL, 0) < 0)
    perror("Setsockopt failed");

```

```

/*
 * Close the socket and exit program.
 */
printf("Closing socket; exiting program...\n");

loop_exit(0, sock);
}
int loop_exit(err, sock)
int err;
int sock;
{
    if (sock > 0)
    {
        if (sclose(sock) < 0)
            perror("Sclose failed");
    }
    exit(err);
}

```

Index

A

- accept, 4–8
- Access control information, 1–15, 1–18
- ADAPTER STATUS, 7–23
- ADD GROUP NAME, 7–27
- ADD NAME, 7–29
- Assembly language interface, 6–1
 - establishing sessions, 6–3
 - how applications communicate, 6–2
 - introduction, 6–1
 - task-to-task, 6–1

B

- bcmp, 5–6
- bcopy, 5–7—5–8
- bind, 4–11
- bzero, 5–8—5–9

C

- C language function calls
 - bcmp, 5–6
 - bcopy, 5–7
 - bzero, 5–8

- dnet_addr, 5–9
- dnet_conn, 5–11
- dnet_eof, 5–16
- dnet_getacc, 5–17
- dnet_getalias, 5–19
- dnet_htoa, 5–20
- dnet_installed, 5–21
- dnet_ntoa, 5–23
- dnet_otoa, 5–24
- dnet_path, 5–25
- getnodeadd, 5–29
- getnodeent, 5–30
- getnodename, 5–33
- nerror, 5–34
- perror, 5–35
- C language subroutines, 5–1
- C programming library
 - creating, 4–1, 5–1
 - DECnet utility function calls for, 5–4
- CALL, 7–31
- CANCEL, 7–33
- CHAIN SEND, 7–35
- Client-server
 - model, 1–12
 - tasks, 1–13, 1–15
- Close a File Handle, 2–13

Close File Handle, 3-9
Command terminal, 1-7
Communications domain, supported by
 DECnet-DOS, A-1
connect, 4-14
Create a File, 2-15
Create/Open a File, 3-10
CTERM. *See* Command terminal

D

Data Access Protocol, error messages,
 G-1
 maccode field values, G-2
 miccode field values, G-3
Data conversions, performing, 2-5
Data members, defined, B-1
Data Structures, NCB, 7-11
Data structures
 access control information for, B-2
 attach, B-3
 defined, B-1
 listen, B-4
 local node information for, B-5
 logical link information for, B-6
 NCB, 7-8, 7-12
 node address for, B-4
 optional user, B-6
 select, B-7
 shutdown, B-8
 socket address, B-8
 socket I/O status, B-9
 socket option, B-9—B-10
 user access control information, B-10
 user-defined buffer, B-11
 user-defined callback routine, B-11
Datalink driver, 1-6

DECnet
 as a background task, 1-6
 components, 1-2
 definition, 1-2
 software, 1-2
DECnet layers, supported by DECnet-
 DOS, A-2
DECnet utility function calls, for C pro-
 gramming library, 5-4
DECnet-DOS
 components, 1-6
 definition, 1-4
DECnet-DOS Network Process (DNP),
 1-7, 6-3, 7-7
DECnet-DOS programs, transporting to
 other systems, H-1
Delete a File, 2-17
DELETE NAME, 7-37
Digital Network Architecture, 1-2
DLL. *See* Datalink driver
DNA. *See* Digital Network Architecture
dnet_addr, 5-9—5-11
dnet_conn, 5-11
 DECnet objects for, A-2
dnet_eof, 5-16
dnet_getacc, 5-17
dnet_getalias, 5-19—5-20
dnet_htoa, 5-20
dnet_installed, 5-21—5-23
dnet_ntoa, 5-23
dnet_otoa, 5-24
dnet_path, 5-25—5-29
DNP. *See* DECnet-DOS Network Pro-
 cess

E

Error completion codes, summary of,
C-1

Error messages

Data Access Protocol, G-1

 maccode field values, G-2

 miccode field values, G-3

for Transparent File Access, F-1

Establishing sessions, for assembly lan-
guage task-to-task, 6-3

Extended error codes, summary of, D-1

Extended error messages

 disconnecting logical link, D-4

 unable to make connection, D-1—D-3

F

File attributes, 2-4

 data type, 2-4

 file organization, 2-4

 fixed header size, 2-5

 maximum record size, 2-5

 record attributes, 2-5

 record format, 2-4

File characteristics, 2-3

File specification, syntax, 2-8

Find First Matching File, 2-18

FIND NAME, 7-39

Find Next Matching File, 2-20

Flag options, for sockets, A-6

Function requests, issuing, 2-10

G

getnodeadd, 5-29

getnodeent, 5-30

getnodename, 5-33

getpeername, 4-17

getsockname, 4-19

getsockopt, 4-21

H

HANG UP, 7-41

I

I/O operations

 blocking, 1-18

 nonblocking, 1-18

Incoming connection requests, maxi-
mum number of, A-7

Intercept routine, for MS-DOS task-to-
task, 3-7

Interrupt vectors, A-10

L

LAT. *See* Local Area Transport

LISTEN, 7-43

listen, 4-27

Load and Execute a Program, 2-22

Local Area Transport (LAT), 1-7

Logical link

 closing, 1-21

 establishing, 1-15

Logical link states, A-7

M

MS-DOS task-to-task, intercept routine,
3-7

N

nerror, 5-34—5-35

NETBIOS

- accessing services, 7-8
- adapter name, 7-6
- asynchronous mode, 7-15
- communication between applications, 7-5
- extended commands, 7-63—7-78
- extended functions, 7-18
- installation checks, 7-9
- issuing calls, 7-11
- LAN support, 7-4
- NCB commands, 7-14, 7-22—7-63
- return codes, 7-11, 7-18
- session control block fields, 7-20
- switches, 7-7—7-8
- synchronous mode, 7-16
- VAX communications support, 7-5
- WAN support, 7-4

NETBIOS commands, 7-17

- ADAPTER STATUS, 7-17, 7-23
- ADD GROUP NAME, 7-17, 7-27
- ADD NAME, 7-17, 7-29
- ADD REMOTE NAME, 7-64
- ADD SERVER NAME, 7-65
- CALL, 7-17, 7-31
- CANCEL, 7-17, 7-33
- CHAIN CALL, 7-17
- CHAIN SEND, 7-35
- DELETE ALL REMOTE NAMES, 7-66
- DELETE ALL SERVER NAMES, 7-67
- DELETE NAME, 7-17, 7-37
- DELETE REMOTE NAME, 7-68
- DELETE SERVER GIVEN NODE NAME, 7-69

- DELETE SERVER GIVEN NODE NUMBER, 7-70

- FIND NAME, 7-17, 7-39

- HANG UP, 7-17, 7-41

- INSTALLATION CHECK, 7-74

- LISTEN, 7-17, 7-43

- READ ALL REMOTE NAMES, 7-71

- READ NUMBER OF REMOTE NAMES, 7-73

- READ REMOTE NAME, 7-75

- READ SERVER BY INDEX, 7-76

- READ SERVER GIVEN NODE NAME, 7-77

- READ SERVER GIVEN NODE NUMBER, 7-78

- RECEIVE, 7-17, 7-45

- RECEIVE ANY, 7-17, 7-47

- RECEIVE BROADCAST DATAGRAM, 7-17, 7-49

- RECEIVE DATAGRAM, 7-17, 7-51

- RESET, 7-17, 7-53

- SEND, 7-17, 7-55

- SEND BROADCAST DATAGRAM, 7-17, 7-57

- SEND DATAGRAM, 7-17, 7-59

- SESSION STATUS, 7-17, 7-61

- NETBIOS emulation, 1-11, 7-1—7-78

- Network activity, terminating, 1-21

- Network Control Blocks (NCB), 7-11, 7-12

- Network management, 1-3

- Network object numbers, 1-18

- Network process

- accessing services, 6-3

- calling, 6-5

- Network programming

- concepts, 1-12

- considerations, 1-12

- Network services, 1-6
 - installation check, 6-4
- Network task specifications, 3-4
 - node, 3-5
 - task, 3-6
- Node addresses, 1-18
- Node names, 1-18
- Normal data messages
 - receiving, 1-20
 - sending, 1-19

O

- Open a File, 2-24
- Open systems interconnect
 - ISO model, 1-2
 - structure for use with DNA, 1-2
- Out-of-band messages, 1-21

P

- perror, 5-35
- Programming considerations
 - DECnet-DOS, 4-2, 5-2—5-4
 - transparent file access, 2-12
 - transparent task-to-task, 3-7
- Programming examples
 - client task, I-1
 - client task-to-task, I-14—I-21
 - client transparent task-to-task,
 - I-6—I-8
 - server task, I-9—I-13
- Programming interfaces
 - assembly language, 1-7

- assembly language task-to-task, 1-11,
 - 6-1
- C language subroutines, 5-1
- C language task-to-task, 1-10, 4-1
- creating the C library for, 4-1
- CTERM, 1-7, 1-12
- LAT, 1-7, 1-12
- NETBIOS, 1-7
- nontransparent task-to-task (C), 1-7
- resident, 1-9
- selecting, 1-8
- transparent file access, 1-7, 2-1
- transparent task-to-task, 1-7, 3-1

R

- Read from a file, 3-12
- Read from a File/Device, 2-26
- Real-time scheduler, 1-6
- RECEIVE, 7-45
- RECEIVE ANY, 7-47
- RECEIVE BROADCAST DATA-
 - GRAM, 7-49
- RECEIVE DATAGRAM, 7-51
- recv, 4-29
- Remote file access, 1-3
- Remote files, accessing, 2-2
- Remote input files, converting, 2-6
 - ASCII, 2-7
 - binary, 2-6
- Remote output files, converting, 2-7
 - ASCII, 2-7
 - image, 2-8
- RESET, 7-53
- Resource sharing, 1-3

S

SCH. *See* Real-time scheduler
sclose, 4-33
select, 4-35
SEND, 7-55
send, 4-39
SEND BROADCAST DATAGRAM,
 7-57
SEND DATAGRAM, 7-59
Session layer, NETBIOS, 7-1
SESSION STATUS, 7-61
setsockopt, 4-43—4-51
shutdown, 4-51—4-53
sioclt, 4-53—4-55
socket, 4-55
Socket interface
 call descriptions, 4-4
 example calling sequence, 4-7
 function calls, 4-5
 SYNTAX section, 4-5
Sockets, 1-13
 communications domain, A-1
 DECnet objects for dnet_conn, A-2
 DECnet options, A-3—A-5
 definitions, A-1
 flag options, A-6
 incoming connection requests, A-7
 interface options, A-8
 logical link states, A-7
 sequenced packet, 1-13
 software modules
 interrupt vectors for, A-10
 TLA strings for, A-9
 stream, 1-13
 supported DECnet layers, A-2
 types, A-9
Software modules, defined for sockets,
 A-9

sread, 4-57
swrite, 4-60

T

Task names, 1-18
Task-to-task communication, 1-8
 functions, 1-3
 nontransparent, 1-9
 transparent, 1-9
Terminal communication, 1-3
TFA. *See* Transparent File Access
TNT. *See* Transparent Network Task
Transparent communication
 client and server functions, 3-1—3-3
 creating a task, 3-4
 exchanging data messages, 3-3
 handshaking sequence
 client task, 3-2
 server task, 3-2
 logical link
 initiating connection, 3-2
 terminating, 3-3
 task-to-task, 3-1
Transparent File Access, 1-10
 command line, 2-2
 displaying status, E-1
 error messages, F-1
 function requests
 Close a File Handle, 2-13
 Create a File, 2-15
 Delete a File, 2-17
 Find First Matching File, 2-18
 Find Next Matching File, 2-20
 Load and Execute a Program, 2-22
 Open a File, 2-24
 Read from a File/Device, 2-26
 Write to a Remote File, 2-28
 introduction, 2-1

- on-line help for, E-3
- removing from memory, E-2
- services, 2-8
- Transparent Network Task
 - control utility, E-1
 - displaying status for TFA, E-1—E-2
 - displaying status for TTT, E-1—E-2
 - removing TFA from memory,
 - E-2—E-3
 - removing TTT from memory,
 - E-2—E-3
- Transparent Task-to-Task, 1-10
 - displaying status, E-1
 - function requests, 3-8
 - close file handle, 3-9
 - create/open a file, 3-10
 - read from a file, 3-12
 - write to a file, 3-14
- on-line help for, E-3

- programming interface, 3-1
- removing from memory, E-2
- Transporting programs, DECnet-DOS to
 - DECnet-ULTRIX, H-1
- TTT. *See* Transparent Task-to-Task

U

- User data, optional, 1-19

V

- Virtual devices, 1-3

W

- Write to a File, 3-14
- Write to a Remote File, 2-28

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA
call 800-DIGITAL

In Canada
call 800-267-6215

In New Hampshire
Alaska or Hawaii
call 603-884-6660

In Puerto Rico
call 809-754-7575

ELECTRONIC ORDERS (U.S. ONLY)

Dial 800-DEC-DEMO with any VT100 or VT200
compatible terminal and a 1200 baud modem.
If you need assistance, call 1-800-DIGITAL.

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: A&SG Business Manager

INTERNATIONAL

DIGITAL
EQUIPMENT CORPORATION
A&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC),
Digital Equipment Corporation, Westminister, Massachusetts 01473

* Any prepaid order from Puerto Rico must be placed
with the Local Digital Subsidiary:
809-754-7575 x2012

FROM: [illegible]

TO: [illegible]

DATE: [illegible]

TIME: [illegible]

PLACE: [illegible]

SUBJECT: [illegible]

REFERENCE: [illegible]

REMARKS: [illegible]

APPROVED: [illegible]

SIGNATURE: [illegible]

POSITION: [illegible]

UNIT: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

REMARKS: [illegible]

READER'S COMMENTS

What do you think of this manual? Your comments and suggestions will help us to improve the quality and usefulness of our publications.

Please rate this manual:

	Poor				Excellent
Accuracy	1	2	3	4	5
Readability	1	2	3	4	5
Examples	1	2	3	4	5
Organization	1	2	3	4	5
Completeness	1	2	3	4	5

Did you find errors in this manual? If so, please specify the error(s) and page number(s).

General comments:

Suggestions for improvement:

Name _____ Date _____

Title _____ Department _____

Company _____ Street _____

City _____ State/Country _____ Zip Code _____

DO NOT CUT - FOLD HERE AND TAPE



NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY LABEL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

digital™

**Networks and
Communications Publications**

550 King Street
Littleton, MA 01460-1289



DO NOT CUT - FOLD HERE

digital